Intelligent Adversary Searches

By

Patrick Pantel

Artificial Intelligence
December 1997

DEPARTMENT OF COMPUTER SCIENCE
The University Of Manitoba
Winnipeg, Manitoba, Canada

Intelligent Adversary Searches

Patrick Pantel, University of Manitoba, Patrick_Pantel@umanitoba.ca

University of Manitoba,
Department of Computer Science,
545 Machray Hall,
Winnipeg, Manitoba
Canada R3T 2N2

Abstract

This paper will provide a theoretical and practical comparison of various adversary searches attempting to play two-person perfect information zero-sum games cleverly. Several different intelligent adversary searches will be described. The best-first searches that I will introduce are SSS* and MT-based algorithms. I will also study several depth-first search algorithms including MiniMax, Solve, Alpha-Beta Pruning, and Scout. The latter algorithms will also be compared in practice on a game called *Pear* (based on Connect Four). I have found that their practical results match very closely to their theoretical analysis. Also, I will present the results of a tournament where each player plays against the others. Scout and Alpha-Beta Pruning are the best algorithms in the game *Pear*. It will also be shown that MT-based algorithms are the best algorithms described in this paper. They are the easiest to understand, the most efficient, and they have modest memory requirements.

ABSTRACT .... **I**

1. INTRODUCTION **1**

1.1. WHAT IS A GAME? 1
1.2. TWO-PERSON PERFECT INFORMATION ZERO-SUM GAME 2
1.3. COMPLEXITY OF GAMES 2
1.3.1. An Example… Tic-Tac-Toe 3
1.3.2. How Do Humans Succeed So Well? 3

2. BASIC ADVERSARY SEARCH ALGORITHMS **5**

2.1. MINIMAX 5
2.1.1. Description 5
2.1.2. Depth Bound 6
2.1.3. Complexity 7
2.2. SOLVE 7

3. ADVANCED ADVERSARY SEARCH ALGORITHMS **9**

3.1. ALPHA-BETA PRUNING 9
3.1.1. Description 9
3.1.2. Complexity 11
3.2. SCOUT 12
3.2.1. Motivation 12
3.2.2. Description 14
3.2.3. Complexity 15
3.3. SSS^* 16
3.3.1. Informal Description of SSS^* *16*
3.3.2. Fixing the space complexity of SSS^* *17*
3.4. MT ALGORITHMS 17
3.4.1. Description 18
3.4.2. MTD-f 20

4. PEAR **23**

4.1. WHAT IS PEAR 23
4.1.1. Rules 23
4.1.2. Complexity 24
4.2. STRATEGY 24
4.3. BOARD EVALUATION 27

5. INTELLIGENT PLAYERS IN *PEAR* **29**

5.1. MAXIMILIAN THE MINIMAX PLAYER 29
5.2. SOLOMON THE SOLVE PLAYER 31
5.3. ANNABELLE THE ALPHA-BETA PLAYER 34
5.4. SCOTT THE SCOUT PLAYER 37

6. RESULTS IN *PEAR* **41**

6.1. TIME COMPLEXITY 41
6.2. EXPANDED NODES 42
6.3. TOURNAMENT 44
6.3.1. Rules 45
6.3.2. Results 46

7. CONCLUSION **49**

REFERENCES **51**

INDEX **53**

1. Introduction

Games have always been a passion of humans. They have historically been perceived as intellectual hobbies. So, it is only natural to view the automation of intelligent game players as a domain of Artificial Intelligence (AI). However, there are many different types of games and each will need a different approach when creating intelligent players. I will concentrate on only one type of game called two-person perfect information zero-sum. Such problems have generated the most attention from AI over the past few decades. The complexities of such games require intelligent searches in order to solve problems efficiently.

1.1. What is a Game?

We have all played some kind of game. Some common examples include tic-tac-toe, Connect Four, poker, backgammon, Risk, Go, checkers, chess, and many more. **Game theory** is the science that is concerned with classifying the different types of games. Every **game** can be defined by the six rules shown in Figure 1.1[Rapoport, 1966].

1. *There must be at least two players;*
2. *The game begins by one or more players making a choice among a number of specified alternative;.*
3. *After the first move is made, a certain situation results. This situation determines who is to make the next move and which are his (or their) alternative;.*
4. *The choices made by the players may or may not become known;*
5. *If a game is described in terms of successive moves, then there is a termination rule.*
6. *Every completed game ends in a certain situation. Every bona fide player receives a payoff.*

Figure 1.1 – The six rules which define a game in game theory [Rapoport, 1966].

The first three rules are very straightforward. However, the fourth rule should be clarified. If all the players know all the alternatives available to a particular player at any point in time, then we have a **perfect information** game. Otherwise, we do not always know all the possible situations for the next ply. A **ply** is simply a single player's action. A **move** is a complete round where all the players have made a single action. Hence, two player games have two plies per move. An example of a game where there is unknown information is Poker. This game does not have perfect information because a player never knows which cards his opponents have (assuming he does not cheat).

1.2. Two-Person Perfect Information Zero-Sum Game

From here on, I will talk only about a specific class of games: **two-person perfect information zero-sum games**. These are games as defined in section 1.1 with the following restrictions: [Von Newmann, 1944]:

1. There are exactly two players;
2. We have perfect information;
3. The payoff is complete.

Assume that player *A* has a payoff of *p*. Then, if the payoff is **complete**, player *B* will have a payoff of *-p*. In games where the payoff is not complete, it may be advantageous to a player to accept a negligible loss if he is able to win more the next time around. It becomes interesting when negotiations are allowed. The Prisoner's Dilemma is an example of such a game. It is very well explained in [Hofstadter, 1985].

1.3. Complexity of Games

Typically, strategy games are extraordinarily complex. Their **search space** (Figure 1.2) grows extremely quickly. More precisely, they grow exponentially. We are therefore dealing with **intractable** problems (Figure 1.2), also known as **hard** problems.

Definition:	*The* ***search space*** *of a game is the set of all possible situations a game could ever be in.*
Definition:	*A problem is said to be* ***intractable*** *or* ***hard*** *if and only if no algorithm with polynomial time complexity may solve it.*

Figure 1.2 – Definition of search space and intractable problems.

1.3.1. An Example… Tic-Tac-Toe

One of the simplest of these games is **tic-tac-toe**. Once a player gains experience, this game becomes very easy. Any descent player should never lose at this game. However, it is remarkable how many different possible games may be played.

Initially, Player *A* may place an *X* in nine different places. Then, Player *B* has a choice between eight squares (since one is now covered with an *X*). Then, *A* has a choice between seven squares and so on until all squares have been filled. Thus, there are $9 \times 8 \times 7 \times \ldots \times 3 \times 2 \times 1 = 9!$ different outcomes of the game. That comes out to 362,880 different possible games! This is in fact an intractable problem albeit of small scale.

1.3.2. How Do Humans Succeed So Well?

How do humans cope so easily with all these possible outcomes? Easily, we reject a huge amount of the possible outcomes based on patterns that we recognise from having seen before. Thus, we do not even consider them. Secondly, we may take advantage of the fact that the board is entirely symmetrical. This then eliminates a great deal of **configurations** (Figure 1.3) to consider.

Definition: *A **configuration** is a problems state at any point in time.*

Definition: *The **initial state** or **initial configuration** is the problem configuration before any operators have been applied.*

Figure 1.3 – Definition of a configuration and an initial state.

Figure 1.4 shows four possible board positions. Assume that it is always *X*'s turn to move. In a), *X* should make a move that grants him the most options for subsequent plies. Therefore, he should choose square [2,2]. As for b), *X* must obviously play in square [1,3] to avoid losing. Similarly, in c), *X* should clearly play in square [3,2] to win the game. Finally, in d), *X* can trap *O* by placing a piece in square [1,3]. He is ultimately setting himself up for a win.

The key to our capability of solving such problems is our ability to **reduce the amount of possible configurations**. Therefore, in order to automate the process of efficiently solving such problems, our adversary searches must somehow reduce the search space.

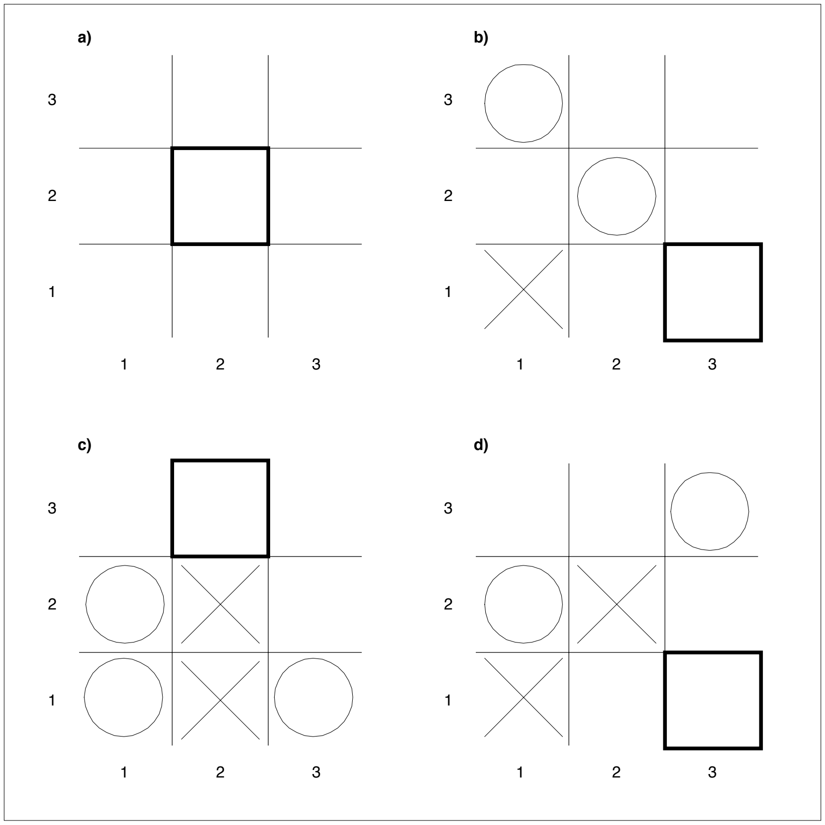

Figure 1.4 – Four possible board positions where it is *X*'s turn. Bold squares indicate where *X* should move. **a)** *X* should place a piece in square [2,2] since this is the square that leaves *X* the most possible options later; **b)** *X* should place a piece in [1,3] to avoid losing to *O*; **c)** *X* should place a piece in [3,2] to win the game; **d)** *X* should place a piece in [1,3] to trap *O*.

2. Basic Adversary Search Algorithms

In the previous section, we have seen that the search space for games is huge. This section will introduce some basic adversary search algorithms for creating intelligent automated players. The two algorithms that we will study are MiniMax and Solve. Both are from the family of **depth-first searches**.

2.1. MiniMax

The **MiniMax** algorithm is used to select the best possible move at any point in time for a player. It is therefore an optimal adversary search algorithm. An **optimal** game search is an algorithm that always returns the best possible move. For the purpose of describing this algorithm, assume Player A is called MAX and Player B is called MIN. Also, assume that it is MAX's turn to move.

2.1.1. Description

Given a current board configuration, MiniMax will logically build a **tree** representing the search space. Then, the algorithm will search through the tree and assign a score to the root node of the tree. This score is the best score MAX could achieve. However, how do we find the best score?

MAX must make his move from the current board configuration. Obviously, he will try to make the best possible move. That is, he wishes to **maximize** his play by selecting the best move (hence the name MAX). Once he makes a move, it will be MIN's turn. MIN's strategy will be to make the best possible move for himself, thus **minimizing** the score for MAX (hence the name MIN).

Now we are ready to describe the algorithm. For the complete algorithm, see Figure 2.1 [UGAI, 1994]. This is a naturally recursive algorithm. Say we call the MiniMax algorithm from the board configuration *N* and we are trying to find a score for MAX. If *N* passes the **termination function** (i.e. the game is over), then we return the estimated score of the board. Otherwise, if *N* represents MIN's turn to move, then MIN wishes to minimize MAX's play. So, the score for *N* would be the minimum of the scores of *N*'s **children** (i.e. all possible moves made from N). In fact, we are returning the minimum of the scores of the MINIMAX value of *N*'s children. Similarly, if *N* represents MAX's turn, then MAX wishes to maximize his play. Thus, the score for *N* will be the maximum of the scores of the MINIMAX value of *N*'s children.

```
function MINIMAX(N) is
begin
  if (N is a leaf) or (depth = DEPTH_BOUND) then
    return the score of this leaf
  else
    Let N1, N2, ..., Nm be the successors of N;
    if N is a Min node then
      return min{MINIMAX(N1), ..., MINIMAX(Nm)}
    else
      return max{MINIMAX(N1), ..., MINIMAX(Nm)}
end MINIMAX;
```

Figure 2.1 – The MiniMax algorithm [UGAI, 1994].

2.1.2. Depth Bound

In any interesting game, it would be impossible to search the entire game tree in a reasonable amount of time. Therefore, we add a condition to MiniMax to recurse until either it reaches a **frontier node** (i.e. a node in the tree that does not have any children) or until it reaches a **depth bound**. Thus, we expand fewer nodes. But, we do not necessarily get the best answer. However, the algorithm is still optimal since it returns the best score for the depth allowed.

Say we assign a value x to a node at depth d. It is possible that if we expanded to one more level, depth $d + 1$, we would have a situation resulting in a very different score than x. This is because the score that we assign to the node at depth d is an approximation of its real value. Unfortunately, it is always possible to look one ply further and find a very different score. This is known as the **horizon problem**.

How do we select a depth bound? This is a very difficult question to answer. Because of the horizon problem, it would be ideal to have a dynamic depth bound. That is, a depth bound that would detect whether or not it should look one ply ahead. Such depth bounds are based on the **quiescence** of a board configuration [Russell, 1995]. This is an estimation on how quiet we believe a configuration to be. A **quiet configuration** is one whose score will not change drastically by looking at one more ply. If we do not consider quiescence, then we can set the depth bound to a static value. A reasonable choice is the maximum possible where the algorithm will yield a result in the required (or acceptable) time.

Unfortunately, this leads us to another problem. If we do not recurse all the way to a frontier node, then how do we assign a value to the board? Well, we cannot assign an exact value, but we can assign an estimate of the board value. This **estimate** is generated from a **board evaluation function**. These functions are specific to each game so we will not discuss examples here. In section 4, we will introduce a board evaluation for the game *Pear*.

2.1.3. Complexity

Let b denote the branching factor of MiniMax and d the maximum depth to which it may recurse. For every node at each level in the tree, MiniMax will expand b nodes. Thus, this algorithm is of order $O(b^d)$. This is therefore an intractable problem. In fact, all adversary search algorithms are intractable. What AI tries to do is reduce the amount of unnecessary steps (i.e. the number of nodes expanded). MiniMax is the worst game-playing algorithm since it expands every node in the search tree. However, it is a vital algorithm since many of the other game search algorithms are based on it.

2.2. Solve

The biggest problem with the MiniMax algorithm is that it expands every node in the search space. The goal of adversary searches is to reduce the amount of nodes expanded. **Solve** is a basic extension of the MiniMax algorithm. It reduces the number of nodes expanded by MiniMax by introducing a minor modification to the original algorithm.

Solve adds an optimisation to the MiniMax algorithm [Pearl, 1984]. When evaluating the scores of the children of N, if N is a MAX node and one of its children has a winning score, then we can immediately give a winning score for N without evaluating any more of its children. This is obvious since we know that no other children of N could give a better score. Similarly, if N is a MIN node and one of its children is evaluated to be a loss for MAX, then we will assign that score to N and stop evaluating the rest of its children.

When we extend the MiniMax algorithm to include this optimisation, we then get the Solve algorithm. The complete algorithm is shown in Figure 2.2.

```
function Solve(N) is
begin
  if (N is a leaf) or (depth = DEPTH_BOUND) then
    return the estimated score of this leaf
  else
    Let N1, N2, ..., Nm be the successors of N;
    if N is a Min node then
      return min_stop_at_MAXMIN{Solve(N1), ..., Solve(Nm)}
    else
      return max_stop_at_MAXMAX{Solve(N1), ..., Solve(Nm)}
end Solve;
```

Figure 2.2 – The Solve algorithm.

Unfortunately, like MiniMax, this algorithm is of order O(b^d). That is, for a tree with branching factor b and of depth d, Solve will expand on the order of O(b^d) nodes. Solve is still a much more interesting player. This is because it reduces the number of expanded nodes. However, it reduces so little that it shares the same complexity as MiniMax. Therefore, like MiniMax, Solve cannot look many plies ahead to get a solution in a reasonable amount of time. In order to search more efficiently (i.e. deeper into the tree), we must expand fewer nodes.

3. Advanced Adversary Search Algorithms

Solve and MiniMax are the most basic intelligent game playing algorithm. However, they do introduce a very important fact. The key to building a good intelligent game player is to make it avoid unnecessary search. Solve does avoid a little search when it finds a child with a winning or losing score. However, we can do much better than that. When we avoid expanding unnecessary nodes, we may evaluate more plies (have a higher depth bound) and therefore make better choices. The algorithms that we will discuss are Alpha-Beta Pruning, Scout, SSS*, and MT-based algorithms. Alpha-Beta Pruning and Scout are both depth-first adversary search algorithms while the others are best-first adversary search algorithm.

3.1. Alpha-Beta Pruning

Alpha-Beta Pruning (α-β) is based on the MiniMax algorithm and is by far the most popular intelligent game playing algorithm where we do not lose any information. That is, the nodes that it does not expand are known not to ever possibly influence the MiniMax value of the root of the tree. Thus, α-β is an optimal algorithm. Since it is based on MiniMax, it is a depth-first game search algorithm.

3.1.1. Description

The idea behind α-β is simple. Say that MIN must choose between *n* moves and it has found that move *i* gives a score *x*. Then, if at any point it knows that node *j* could never have a score smaller than *x*, then it can stop investigating branch *j*. Similarly for MAX, if at any point it knows that node *j* could never have a score greater than *x*, then it can also stop expanding nodes in branch *j*. α corresponds to MAX's largest known score to date and β corresponds to MIN's lowest known score to date.

Initially, $\alpha = -\infty$ and $\beta = \infty$. Alpha and Beta thus represent an upper and lower bound respectively for the MiniMax value of the root of the tree. Therefore, it must hold during the whole algorithm that at a node *N*, $\alpha(N) \leq \text{MiniMaxScore}(N) \leq \beta(N)$. Figure 3.1 describes the algorithms as shown in [UGAI, 1994]. Notice that at any node, α never decreases and β never increases.

```
function AlphaBeta(N, A, B) is ;; A is always less than B
begin
  if (N is a leaf) or (depth_bound is reached) then
    return the estimated score of this leaf;
  Set Alpha to A and Beta to B;
  if N is a Min node then
   For each successor Ni of N loop
    Set Beta to Min{Beta, MINIMAX-AB(Ni, Alpha, Beta)};
    When Alpha >= Beta
     Return Alpha endloop;
   Return Beta;
  else
   For each successor Ni of N loop
    Set Alpha to Max{Alpha, MINIMAX-AB(Ni, Alpha, Beta)};
    When Alpha >= Beta
     Return Beta endloop;
   Return Alpha;
end AlphaBeta;
```

Figure 3.1 – The Alpha-Beta Pruning algorithm [UGAI, 1994].

Figure 3.2 shows a sample execution of the algorithm on a three-ply tree with a constant branching factor of two. Part a) shows the full tree. Part b) shows the point where *B* has fully expanded its left child and is about to expand child *E*. At this point, $\alpha = -\infty$ and $\beta = 8$. We are now ready to expand *E*, a MAX node. *E* expands *J* and sets α to 10. Since 10 > 8 (the current β value), we do not need to expand further. This is because MAX could do no worse than 10, but *B* has already the choice of a better score of 8. Therefore, *B* would never choose this branch. So, *E* returns a value of 8. Part c) shows the result of fully expanding *B*.

Now, d) shows the full expansion of *C*. At this point, *F* must be expanded and *L* must be expanded. *L* returns a value of 2 to *F* but since α is already 8, α does not change. *F* then tries to expand *M*. *M* returns 4 but again, since α is already 8, α does not change. So, *F* returns a score of 8 to *C*. *C* sets β to 8 (since $8 < \infty$). Now, since $\alpha \geq \beta$ (i.e. $8 \geq 8$), we know that *C* could never have a better score than 8 so we return a score of 8 to *A*. *A* gets a score of 8 from both its children and therefore the MiniMax value of this tree is 8. Figure 3.3 shows which frontier nodes in this tree have been expanded by Alpha-Beta Pruning.

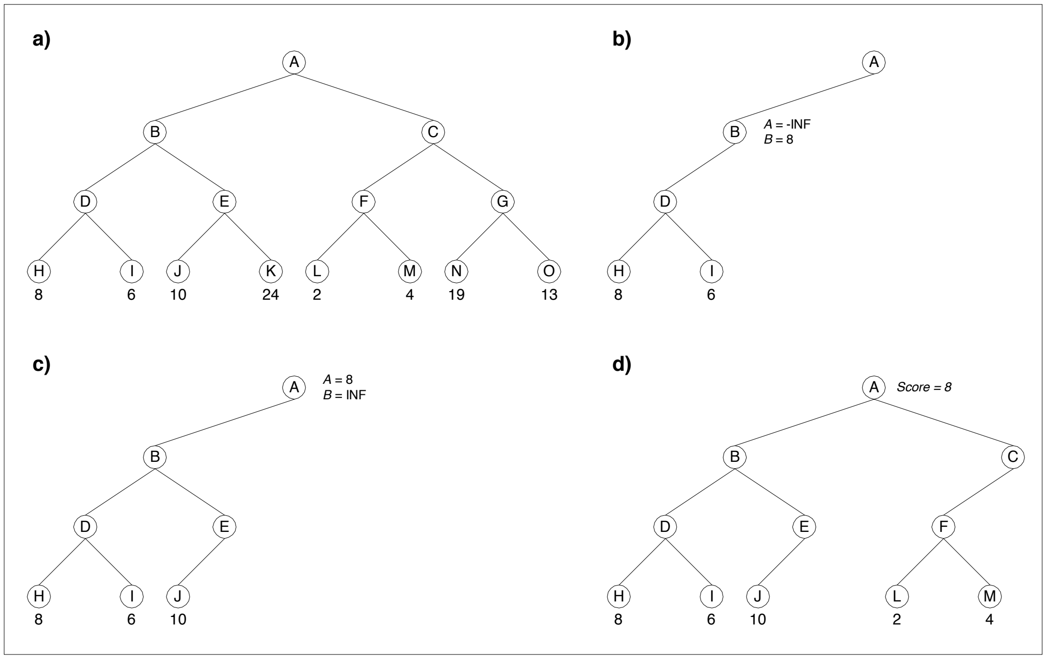

Figure 3.2 – Sample execution of the Alpha-Beta Pruning algorithm. **a)** The complete tree; **b)** Full expansion of the left children of node *B*; **c)** Full expansion of the node *B*; **d)** Finished algorithm showing the score of the root of the node.

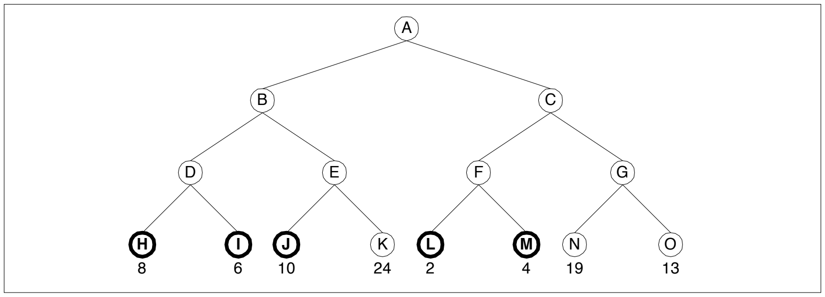

Figure 3.3 – Bold nodes represent frontier nodes expanded by Alpha-Beta.

3.1.2. Complexity

On average, the use of α-β over pure MiniMax reduces the effective branching factor by a factor of two. This is assuming that the best move for each player is always the leftmost node within its siblings. The **effective branching factor** is the branching factor needed for a uniform tree of depth *d* to contain *n* nodes. This would not be so great if α-β did not expand nodes that could influence the MiniMax value of the root of the tree. However, that is not the case. In fact, all nodes that are not expanded by α-β are nodes that could never influence the MiniMax value of the root of the tree [Johnson, 1997]. Therefore, α-β expands O($b^{d/2}$) nodes [Johnson, 1997]. Johnson states that this upper bound is only when d is even. For odd d, α-β effectively expands O($(b+1)^{d/2}$) nodes.

From this, we can conclude that α-β expands only approximately the square root of the number of nodes expanded by Solve. Furthermore, α-β is optimal just like MiniMax and Solve. Therefore, α-β could theoretically solve a problem in the same amount of time as MiniMax and Solve by expanding twice as many levels. That is, if Solve expands to a depth of p plies, then α-β will expand $2p$ plies in the same amount of time.

3.2. Scout

Scout is another depth-first adversary search algorithm that is based on Alpha-Beta Pruning. It was initially created as a "theoretical tool for showing the existence of a strategy that achieves an optimal asymptotic performance" [Pearl, 1984]. However, it has recently been discovered to have practical merit as well. In fact, it performs as well and sometimes better than α-β.

3.2.1. Motivation

The principle behind Solve is rather simple. In Figure 3.4, say that A is a MAX node and that B is estimated to have a score of x. Then, α-β will try to evaluate the score of C and will only prune it once it finds that the estimated value of C is $\leq x$. Scout on the other hand will only evaluate the value of C, known as the ***Eval*** function, if it can affirm that this value will be greater than x. Scout calls this affirmation the ***Test*** function.

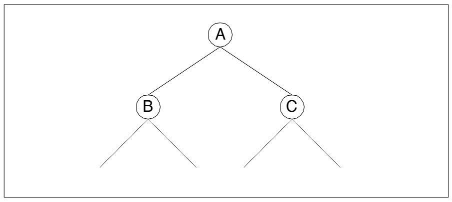

Figure 3.4 – Motivation for the Scout algorithm.

At glance, this seems wasteful. It appears that we will end up duplicating allot of work. That is, if we cannot prove that $C \leq x$, then we will have to re-expand under C to evaluate it. However, it has been shown that most of the time, the estimated value of C will in fact be $\leq x$. Why not just use α-β though? As suggested in [Pearl, 1994], the *Test* function is much quicker than the *Eval* function. This is because "*Test* induces many **cut-offs** not necessarily permitted by *Eval* or any other evaluation scheme".

```
a)
function Test(N, v) is ;; Returns true if the estimated value of $N < v$
begin
  if (N is a leaf) or (depth_bound is reached) then
    if the estimated score of $N > v$ then return true;
    else return false;
  If N is a Max node then
    For each successor Ni of N loop
      if Test(Ni, v) then return true;
    return false;
  else
    For each successor Ni of N loop
      if not Test(Ni, v) then return false;
    return true;
end Test;

b)
function Eval(N) is ;; Returns an estimate of the value of N
begin
  if (N is a leaf) or (depth_bound is reached) then
    return the estimated score of N
  if N is a Max node then
    Set Value to Eval(N1) ;; N1 is the leftmost child of N
    For each successor Ni of N except N1 loop
      if Test(Ni, Value) then
        Set Value to Eval(Ni)
    return Value;
  else
    Set Value to Eval(N1) ;; N1 is the leftmost child of N
    For each successor Ni of N except N1 loop
      if not Test(Ni, Value) then
        Set Value to Eval(Ni)
    return Value;
end Eval;
```

Figure 3.5 – **a)** The *Test* function from Scout; **b)** The *Eval* function from Scout.

Once we know that the estimated value of one child *C* of a MAX node is greater than *x*, then all the other children from MAX may be ignored. Alpha-Beta, on the other hand, would require us to continue checking the other children to detect if any of them had an even better score. It has been shown in [Pearl, 1984] that despite some duplicated expansions, "Scout achieves optimal performance for deep trees".

3.2.2. Description

As introduced above, Scout is based on two functions: *Test* and *Eval*. Figure 3.5 describes both of them [Pearl, 1984]. Figure 3.6 gives an example of the execution of the Scout algorithm.

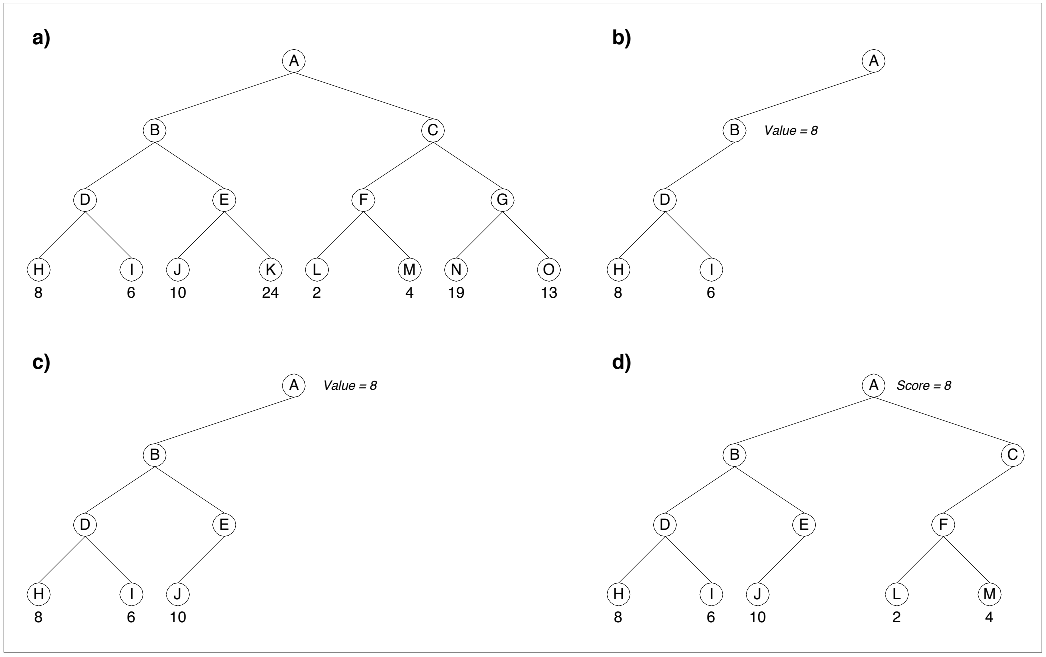

Figure 3.6 – Sample execution of the Scout algorithm. **a)** The complete tree; **b)** Full expansion of the left children of node *B*; **c)** Full expansion of the node *B*; **d)** Finished algorithm showing the score of the root node.

Part a) shows the complete tree before the execution. Part b) shows the point where all of *B*'s left children have been expanded and *B* is about to expand child *E*. *Value* has been assigned the score 8 from the subtree with *D* at the root. Now, we must test *E* to see if its estimated value is smaller or equal to 8. So, we call *Test*(*E*, 8). We test *E*'s first child, *J*. Its estimated value is 10. Since 10 > 8, *Test*(*J*, 8) returns true. Therefore, *Test*(*E*, 8) returns true. Since *Test*(*E*, 8) returned true, the estimated value of *E* is greater than 8. Therefore, we do not need to evaluate *E*. Since we have also finished evaluating all the children of *B*, *B* returns a score of 8 to *A* (*Value* is set to 8 in the *A* block). This scenario is show in c).

Now, *A* must test if the estimated value of *C* is greater than 8. So, it calls *Test*(*C*, 8). This in turn recurses to call *Test*(*F*, 8) which recurses to *Test*(*L*, 8). Since 2 ≤ 8, *Test*(*L*, 8) returns false. Then, *Test*(*F*, 8) must recurse to test the value of its other child, *M*. So, it calls *Test*(*M*, 8). Again, 4 ≤ 8 so *Test*(*M*, 8) returns false. Since *F* could not find any of its children with a score > 8, *Test*(*F*, 8) must return false to C. Thus *Test*(*C*, 8) also returns false.

Therefore, *A* must not evaluate the value of node *C*. It knows that it could never be greater than 8. So, since *A* has finished testing all its children, it returns the MiniMax value of 8. Part d) shows this result. Figure 3.7 shows all the leaf nodes that were tested or evaluated by Scout.

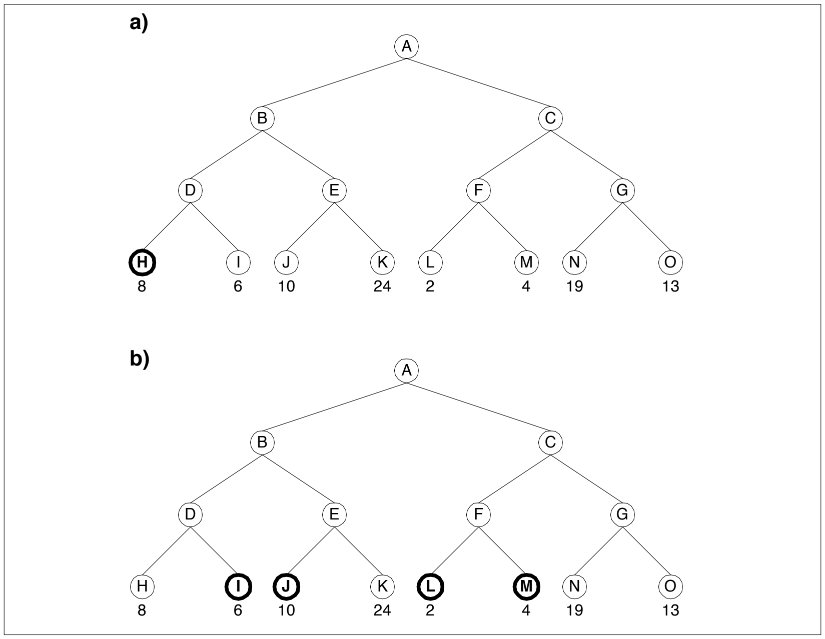

Figure 3.7 – Bold nodes represent: **a)** frontier nodes expanded by the *Eval* function in Scout; **b)** frontier nodes expanded by the *Test* function in Scout.

3.2.3. Complexity

Notice that Scout returned the same answer as α-β. Also, in both algorithms, the same amount of frontier nodes was expanded. The only difference is that α-β evaluated all of them while Scout evaluated only one and tested the other four. It has been proven that for deep subtrees in games with continuous scores, Scout expands the same order of nodes as Alpha-Beta Pruning, $O(b^{d/2})$ nodes [Pearl, 1984].

However, for games with discrete scores, Scout in fact expands fewer nodes than Alpha-Beta Pruning [Pearl, 1984].

3.3. SSS*

Solve, Alpha-Beta and Scout are all flavours of depth-first searches. A couple decades ago, an alternative approach was considered. **SSS*** (Stockman, 1979) is a **best-first search** algorithm. It has been proven to be an optimal algorithm, even superior to the wide spread Alpha-Beta Pruning algorithm. By superior, we mean that α-β will never skip any nodes expanded by SSS*, but SSS* will occasionally skip a node expanded by α-β.

The question then arises: Why isn't SSS* as ubiquitous as α-β. Unfortunately, since SSS* is from the family of best-first search algorithms, it suffers similar caveats as Greedy search and A* search (other well known best-first search algorithms). The major disadvantage of best-first search algorithms is their poor **space complexity**. Unfortunately, the memory requirements for SSS* are so large that it becomes impractical to use. In this section, we will give a brief and informal explanation of the SSS* algorithm and give suggestions on how to improve its memory requirements.

3.3.1. Informal Description of SSS*

Like all best-first searches, SSS* will always expand the cluster which it deems to have the best score at any point in time [Pijls, 1995]. A **cluster** is simply a subtree currently being expanded. In the event of a tie, we will always expand the left-most cluster. Note that only frontier nodes contain a numerical score. Its score serves as an upper bound for all clusters that contain it as a sub-tree. Until a cluster has reached a frontier node, we will assign them a value of ∞ since we do not yet have an upper bound. The upper bound values will serve to sort the subtrees. This permits SSS* to always expand the cluster with the best score to date.

A **priority queue** will be used to contain all the subtrees ready for expansion. The enqueue function (i.e. the priority) will simply be the bound of the clusters. Once one solution is fully developed, the path to get to it is the path of the optimal solution and we can assign to the root of the tree the MiniMax value of the frontier node in the solution.

From this, it is clear that SSS^* must "remember" every path it has visited during the algorithm. This is due to the fact that SSS^* jumps around in the tree since it always expands the most promising cluster. Thus, for a tree of depth *d*, with a branching factor of *b*, in the worst case, the **space complexity** of SSS^* is $O(b^d)$! This is terrible comparing to the space complexity of depth-first search algorithms such as Solve, Alpha-Beta, and Scout which are $O(bd)$.

3.3.2. Fixing the space complexity of SSS^*

Obviously, SSS^* is impractical in any useful game because of its space complexity. However, it could be potentially terrific if we could constrain its memory. Since SSS^* is a best-first algorithm, we should try similar techniques used to improve other best-first searches. In order to improve A^*, one suggestion was to combine it with **iterative deepening**. In fact, that was one proposed solution for SSS^*. It is called **Iterative Deepening SSS^* ($IDSSS^*$)**. Unfortunately, it has been shown that $IDSSS^*$ is in fact very slow in execution [Roizen, 1983], as was IDA^*.

Another suggestion follows the idea of the SMA^* algorithm (A^* with memory constraints). It is called **$MemSSS^*$** [AAAI, 1996]. This algorithm limits the number of nodes that may be "remembered" at any point. In addition, its keeps in memory the best forgotten value of the forgotten children of every node in memory.

This approach has been very successful even though it may not always obtain the optimal solution. It will not obtain an optimal solution if the optimal solution is at depth *d* and $MemSSS^*$ has a memory bound smaller than $d + 1$. That is, the memory bound must be large enough to contain the path to the optimal solution. Otherwise, $MemSSS^*$ is not optimal.

3.4. MT Algorithms

As shown in Section 3.3, SSS^* unfortunately has little practical interest. In this section, we introduce a family of algorithms based on the *Test* function from the Scout algorithm. The family is called

Memory-enhanced Test (MT) algorithms. They have been shown to be easier to understand and more efficient than both Alpha-Beta Pruning and SSS*.

3.4.1. Description

It was shown that SSS* can be described as an Alpha-Beta Pruning search with transposition tables [Plaat, 1995]. A **transposition table** is simply a table of memory to store previously visited nodes. Based on the *Test* function from Scout, Plaat introduced a new function called **Memory-enhanced Test (MT)**. Like *Test*, MT is used to traverse a tree in search of an answer to a binary question.

Unfortunately, most evaluation functions in games are not binary. Plaat showed that MT could be modified to create new algorithms that could be used to solve non-binary evaluation functions. One of these modifications became the MTD-*f* algorithm.

```
function MT(n, y) is ;; Returns a bound on the estimated score of node n
begin
  if retrieve(n) = found then
    if n.f- > y then return n.f-;
    if n.f+ < y then return n.f+;
  if (N is a leaf) or (depth_bound is reached) then
    Set n.f- to the estimated score of this leaf;
    Set n.f+ to the estimated score of this leaf;
    store(n);
    return the estimated score of this leaf;
  Set g to -INFINITY;
  For each successor Ni of n loop until g < y
    Set g to max{g, -MT(Ni, -y);
  if g < y then Set n.f+ to g
  else Set n.f- to g
  store(n);
  return g
end MT;
```

Figure 3.8 – The Memory-enhanced Test (MT) algorithm.

Figure 3.8 shows the MT algorithm [Plaat, 1994]. MT will try to prove that $g < \omega$ or that $g \geq \omega$. Its output will be a bound on the score of the node. In order to remove the **asymmetry** between the "<" and the "≥", Plaat suggested that we use as an input parameter $\omega - \varepsilon$ or $\omega + \varepsilon$, where ε is a value smaller than the difference between the value of any two frontier nodes. Now, the function need not worry about the asymmetry. Let y be this parameter. An obvious precondition of this algorithm is that y must be different than all the scores of the frontier nodes of the tree.

Let N be the current node that we are expanded. The first thing that we do in MT is attempt to retrieve the transposition table information. If previously we had searched to our current depth, the transposition table information will be found. In such a case, the search can be cut-off and we return the lower bound if it is greater than y or the upper bound if it is smaller than y. Whenever we have completed the expansion of a node, we store in the transposition table the lower or upper bound of the value. Let g be the bound on the score of the node. The output of MT will be g. Also, if $g < y$, MT will store g as the upper bound in the transposition table. Otherwise, MT will store g as the lower bound in the transposition table.

```
function MTD+∞(n) is ;; Returns the estimated score of node n
begin
  Set g to INFINITY;
  loop until g = bound
    Set bound to g;
    Set g to MT(n, bound - e);
  return g;
end MTD+∞;
```

Figure 3.9 – The MTD+∞ algorithm.

MT only gives us a bound on the estimated score of the node. By using MT as a driver and calling it repetitively, we can use it to home in on the true estimated score of the node. We call such a driver MTD [Plaat, 1994]. Let f be the true estimated score of a node. Figure 3.9 describes the MTD driver that calls MT repetitively to home in on f. Let g represent an upper bound on f, initially set to ∞. The algorithm will continually call MT until g becomes f. We call this algorithm MTD+∞.

If the size of the transposition table is of order $O(\omega^{\lceil d/2})$ and that no collisions occur in the transposition table, MTD+∞ will examine the same leaf nodes as SSS^* and in the same order [Plaat, 1994]. That is, MTD+∞ is SSS^*.

A generalisation of this driver may be created. This generalisation can then be used to create a variety of different algorithms. In order to do this, we must define the variables in an MT driver. There are actually two variables: *first* and *next*. *First* is the initial bound on MT (e.g. ∞ in MTD+∞) and *next* is an operator that sets the bound variable. These can then be used as parameters to our general driver. Figure 3.10 shows the code for a general **MT driver** [Plaat, 1994].

```
function MTD(first, next, n) is ;; Returns the estimated score of node n
local var $f^+$, $f$ ;
begin
  Set $f^+$ to ∞;
  Set $f$ to ∞;
  Set bound to first;
  Set g to first;
  loop until $f^+ = f$
   { The next operator must set the variable bound }
   next;
   Set g to MT(n, bound - e);
   if g < bound then Set $f^+$ to g
   else Set $f$ to g
  return g;
end MTD;
```

Figure 3.10 – Algorithm for a general MT driver.

Now, by calling MTD by setting *first* to ∞ and *next* to g, we get SSS^* (or MTD+∞).

3.4.2. MTD-*f*

We can use MTD to define many different search algorithms. One that has been very successful is **MTD-*f*** [Plaat, 1994]. MTD-*f* has been shown to out-perform Alpha-Beta, Scout, and SSS^*. More importantly, it does so with modest memory requirements, thanks to the MT algorithm.

One problem with MTD+∞ (SSS^*) is that it starts off with an initial bound of ∞. MTD-*f* simply uses the idea that we would avoid plenty of searches by starting at a bound closer to the real score of the root of the tree. By doing so, we would avoid all the initial searches of SSS^* used to get us on the right track.

How do we choose the initial bound? Depending on the game at hand, it is possible to make a guess. This is obviously not the best solution since our guess may in fact be very far from the reality. Another possibility is to use the score of the previous evaluation. This is much better since often the estimated score of a board does not radically change from one move to the next. A similar strategy, used only when iterative deepening is combined with the search, is to assign the bound to the score of the previous iteration.

Since we can easily build many different algorithms using the generalised driver algorithm, we can easily see how similar adversary search algorithms really are. In terms of efficiency, all MTD algorithms are more efficient than pure Alpha-Beta Pruning. Why is Alpha-Beta more popular? It should not be. In fact, MT based algorithms are more efficient, have good space complexity, and are actually easier to understand [Plaat, 1995]! Strangely enough, Alpha-Beta is still the most talked about algorithm.

4. *Pear*

We will compare some of the adversary search algorithms that I have described in sections 2 and 3 on a game called *Pear*. ***Pear*** follows the same rules as the popular game Connect Four. *Pear* is complex enough that it is impossible to consider searching the whole game tree and simple enough that we do not have a high branching factor. Furthermore, the branching factor is relatively constant. I will also introduce in this section the strategy and board evaluation function that will be used for *Pear*. For all figures depicting game situations, light pieces will belong to Player A and dark pieces will belong to Player B.

4.1. What is Pear

I originally designed *Pear* to be an **object oriented** game. However, I have implemented it using Borland Turbo Pascal 3.0 for DOS. It uses a **graphical user interface** where all the control lies on the user. The interface between the user and the game is done with only a mouse. There is no keyboard required. It has been implemented as a research tool and therefore does not offer many help facilities. An executable has been included with this paper permitting the user to play against four different intelligent automated game players. Also, the user may watch different computers play head-to-head. Please note that *Pear* is an MS-DOS application that requires a Microsoft®™ compatible mouse.

4.1.1. Rules

The **rules** for *Pear* are the same as for the game Connect Four. There are two players who alternate turns. The board incorporates six rows and seven columns. A **legal move** consists of placing a piece on any column that has not yet been filled. The piece will ‘slide’ down to the first unoccupied row from the bottom. The game is over when:

1. A player has 4 of his pieces in a row horizontally; or
2. A player has 4 of his pieces in a row vertically; or
3. A player has 4 of his pieces in a row diagonally; or
4. No more legal moves may be made.

In cases 1-3, the player with the four pieces in a row is declared the winner. In case 4, there are no winners (i.e. the game is a draw).

4.1.2. Complexity

It is important to figure out how many possible games may be played out to reinforce the fact that a search algorithm could never search the whole game tree. Since there are $6 \times 7 = 42$ squares on the board, the maximum number of plies for a game is 42. Also, at any configuration in the search space, there will be a maximum of 7 legal moves. In fact, there will almost always be 7 legal moves available until columns get completely filled up. Since we are looking for an upper bound on the number of possible games, we will assume that there are always 7 legal moves available (this is a very good approximation). Then, an upper bound on the number of possible games that may be played out is $7^{42} \approx 3 \times 10^{35}$.

Obviously, no computer has the power to search 3×10^{35} games in any reasonable amount of time. Therefore, this is a great game to use to compare our search algorithms. In order for an adversary search to succeed, it must choose wisely which nodes in the search space that it will expand.

4.2. Strategy

In order to evaluate a given board position (i.e. a board evaluation function), we must define a **strategy** that would yield good moves. How do we define a good move in *Pear*?

All things being equal, we will prefer selecting a move near the **middle of the board**. The middle of the board is the most advantageous position since it is the only column which lets you win on both diagonals and which gives you the most ways of winning horizontally. Figure 4.1 shows all the winning positions from square [3,4]. The closer we get to the side columns, the least number of winning ways exist horizontally.

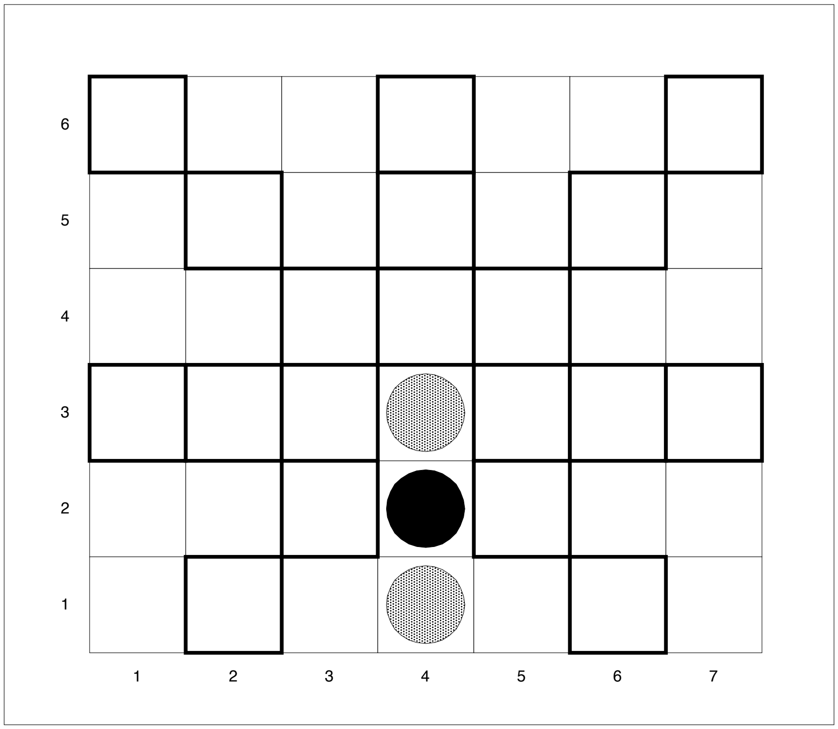

Figure 4.1 – All bold squares represent squares that may participate in a win with square [3,4].

Also, we will prefer moves that yield two same pieces in a row rather than just a single piece. Similarly, we will prefer having three pieces in a row than two pieces in a row. Trivially, the moment we can have four pieces in a row, we take it since this yields a win for us.

Note that the previous strategy is not as good as it might seem. This is because we often have the possibility of choosing a move that results in having two or three pieces in a row. However, often the pattern will never give rise to a win. Figure 4.2 describes this case.

The previous strategies are **offensive strategies**. We must also consider **defensive strategies** to be successful. A simple strategy is to block the opponent from having a good offensive board position. That is, if the opponent has two or three pieces in a row, then we might want to consider blocking him. Figure 4.3 gives an example of this.

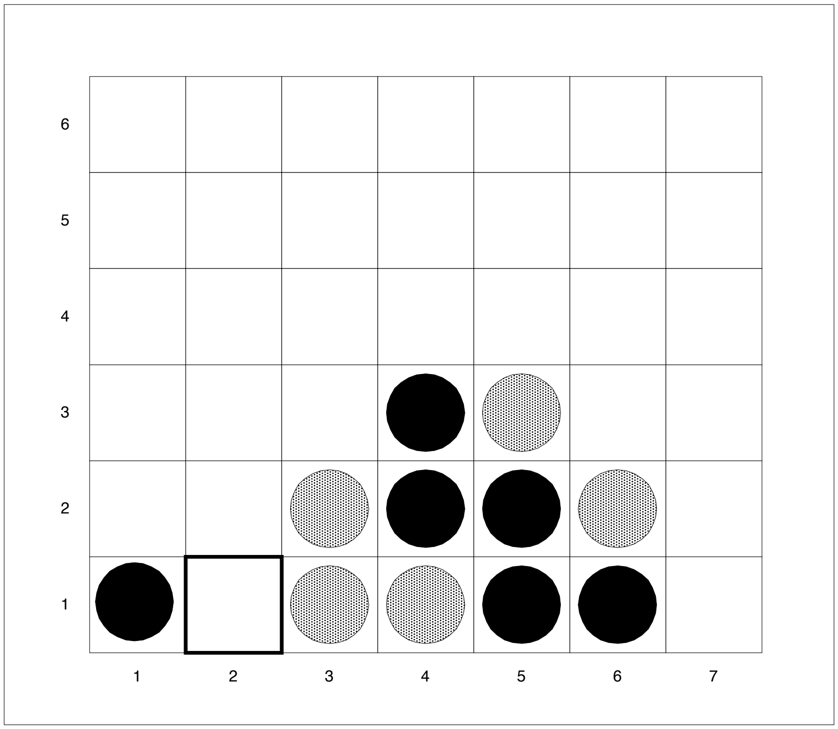
6
5
4
3
2
1
1
2
3
4
5
6
7

Figure 4.2 – Moving to square [1,2] would not be a good strategy. Even though we have three pieces in a row horizontally, we could never place a fourth.

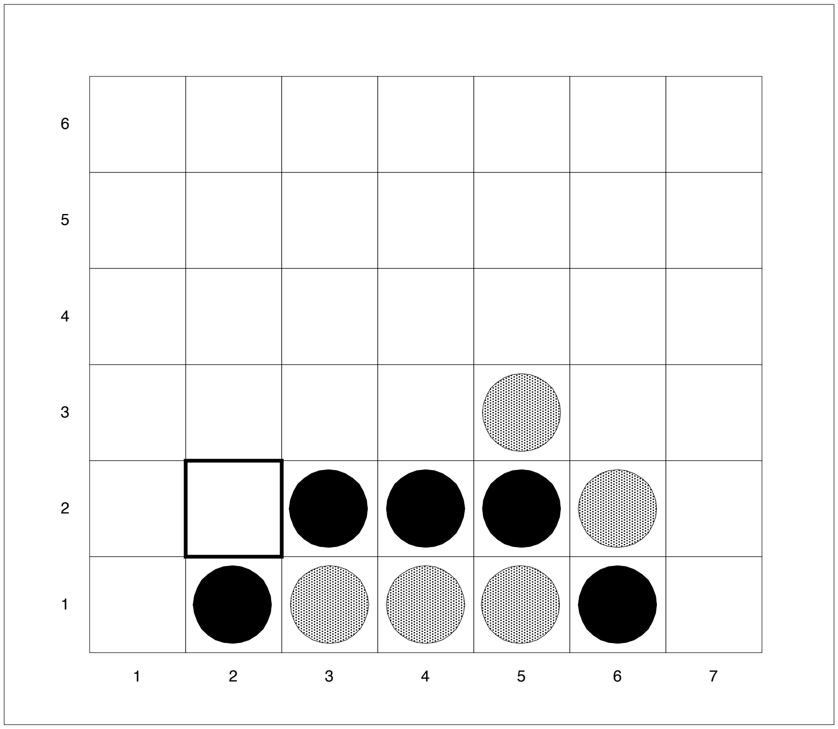
6
5
4
3
2
1
1
2
3
4
5
6
7

Figure 4.3 – Player A should put a piece in square [2,2] to block a win from player B.

4.3. Board Evaluation

These strategies can be used and combined to generate our **board evaluation function**. We need such a function since it would be impossible for any adversary search to traverse the complete game tree. Thus, we need an **estimation** of the value of a board even when the game is not over.

The inputs of the board evaluation function are the current player and the current board configuration. For our description, say it is Player A's move. The board evaluation function will return a score in the range of (-MAXINT, MAXINT]. Figure 4.4 shows a series of board situations with their board evaluation (always assuming we are evaluating for Player A). Here is how we evaluate a board in *Pear*:

For Player A, sum the total of the following scores (maximum MAXINT):

1. For all patterns of four pieces in a row add MAXINT to the score;
2. For all patterns of three pieces in a row add 16 points to the score;
3. For all patterns of two pieces in a row add 4 points to the score;
4. For all single pieces not part of any pattern add 0 points to the score.

Note that a pattern can be horizontal, vertical, or diagonal. Also, a pattern can only be considered once and it will be for the category closest to number 1. This precludes us from taking a three-piece pattern and also counting the two two-piece patterns it forms. Now, we must also consider Player B's pieces.

For Player 2, add to the previous total the sum of the following scores :

1. For all patterns of three pieces in a row add -16 points to the score;
2. For all patterns of two pieces in a row add -4 points to the score;
3. For all single pieces not part of any pattern add 0 points to the score.

Since it is Player A's turn, we do not need to consider the case of patterns with four pieces in a row. This is because such a pattern would have caused a win in the previous move. It is important to note that the board evaluation function in *Pear* is **symmetrical**. That is, given a particular game configuration and a score of x for Player A, then the score for Player B at that same point in time is $-x$.

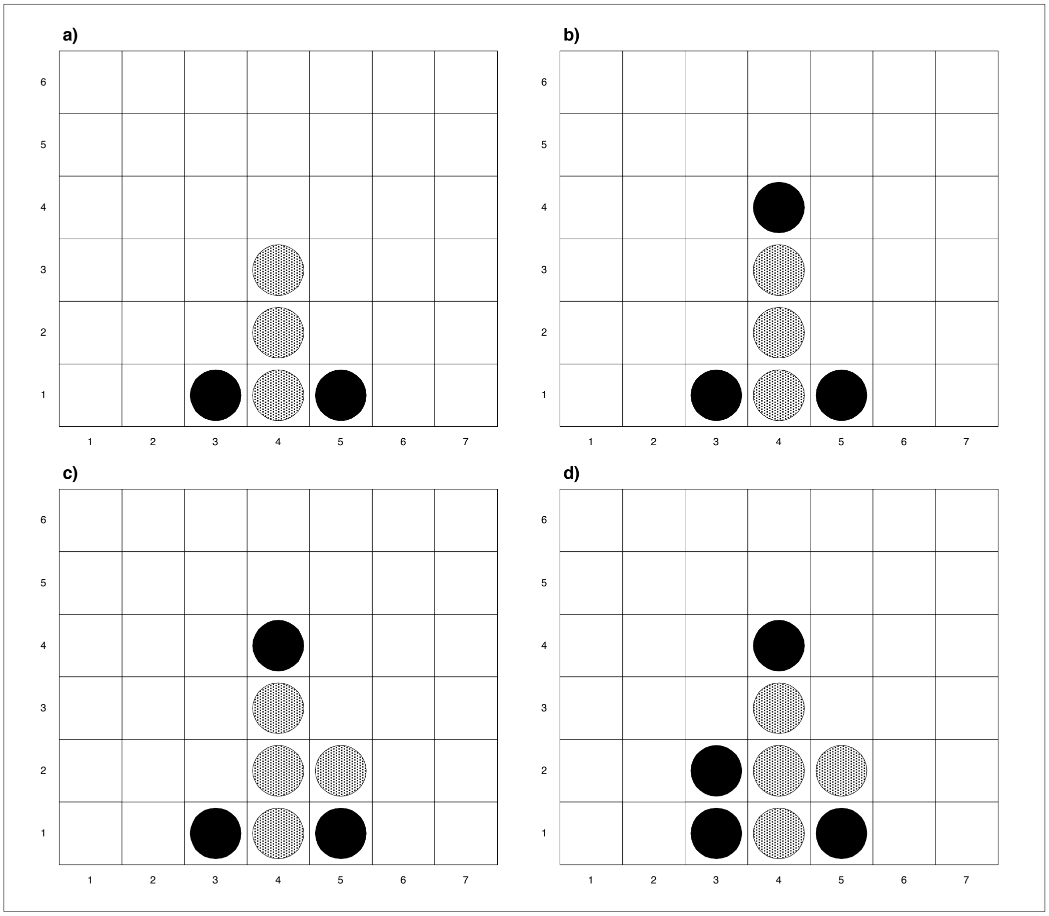

Figure 4.4 – Board evaluation from Player A's point of view. **a)** Score = 16; **b)** Score = 0; **c)** Score = 12; **d)** Score = 8.

In Figure 4.4 a), there is only one pattern and it is for Player A going vertically from [1,4] to [3,4]. Since this pattern could lead to a win, the score is 16. For b), Player B has blocked the pattern for Player A so that it can no longer lead to a win. Therefore, the score is 0. In c), there are three 2 piece patterns for Player A: [1,5] – [2,5] going diagonally, [2,5] – [3,4] going diagonally, and [2,4], - [2,5] going horizontally. Since all these patterns could lead to a win, we have a score of 3 × 4 = 12. In d), Player A has the same number of patterns than in part c). However, Player B now has a 2-piece pattern going vertically from [1,3] to [2,3]. Therefore, we have a score of (3 × 4) + (-4) = 8.

This board evaluation will allow us to estimate the value of a board position. We can then use it to create intelligent adversary searches.

5. Intelligent Players In *Pear*

Now that we have defined a board evaluation function, we can use it to create intelligent adversary searches based on some of the algorithms we have seen in sections 2 and 3. We will create four intelligent adversary searches in *Pear*: Maximilian the MiniMax algorithm, Solomon the Solve algorithm, Annabelle the Alpha-Beta Pruning algorithm, and Scott the Scout algorithm.

All of the algorithms have been implemented to take advantage of the **reversibility** of operations in *Pear*. Every time we recurse in an algorithm, we do not create a whole new board structure. Instead, we always use the same board structure. Before recursing, we add a move to the board and after the recursion is over, we remove that same move. This saves an enormous amount of space and a considerable amount of time (i.e. we do not have to duplicate the board for every call).

Please note that all tests performed where made on a Pentium 133 Mhz machine with 32 MB RAM using Windows 95 as the operating system.

5.1. Maximilian the MiniMax Player

The simplest of all the algorithms that has been implemented is the MiniMax algorithm. In *Pear*, **Maximilian** is the player that uses the MiniMax algorithm. Here is the Pascal algorithm used to implement Maximilian:

```
Function GetMinimaxScore(Level: Integer; Row, Col: Integer; var theMove: Integer): Integer;
  {-----------------------------------------------------------------}
  {-- This function returns the best possible move using the simple --}
  {-- Minimax.                                                      --}
  {-- NOTE: Row, Col are the row and column that were moved to to   --}
  {--       get to this state.                                      --}
  {-----------------------------------------------------------------}
  Var
    i, theLocalScore, theBestMove, theRow, Temp, WhoseTurn: Integer;
    Scores: array[1..Cols] of Integer;
  begin
    {-- Set the WhoseTurn field in the board structure --}
    if Level mod 2 = 1 then WhoseTurn := WhoIsMin
    else WhoseTurn := WhoIsMax;

    {-- If we cannot expand further, return the heuristic (score)   --}
    {-- of the current board.                                       --}
    if level = theMinimaxDepthBound then begin
      theMinimaxBoard^.WhoseTurn := WhoIsMax;
      GetMinimaxScore := h(theMinimaxBoard);
    Exit;
  end;

  {-- If we have generated a goal state return MAX. --}
  {-- If we have generated a tie state return 0.    --}
  if (Level <> 0) then begin
    theMinimaxBoard^.WhoseTurn := WhoseTurn;
    if IsGameOver(theMinimaxBoard, Row, Col, Temp, Temp, Temp) then begin
      if Level mod 2 = 1 then GetMinimaxScore := HEURISTIC_MAX
      else GetMinimaxScore := -HEURISTIC_MAX;
      Exit;
    end;
    if IsTieGame(theMinimaxBoard) then begin
      GetMinimaxScore := 0;
      Exit;
    end;
  end;

  {-- Get the score of all children's nodes. --}
  if Level mod 2 = 1 then begin  {-- MIN Node --}
    for Temp := 1 to COLS do begin
      i := GetNextBestColumn(Temp, i);
      if IsMoveLegal(theMinimaxBoard, i) then begin
        IncMinimaxEvals;
        theMinimaxBoard^.WhoseTurn := WhoseTurn;
        AddPiece(theMinimaxBoard, i, theRow);
        Scores[i] := GetMinimaxScore(Level + 1, theRow, i, theMove) - ColumnValue(i);
        RemovePiece(theMinimaxBoard, theRow, i);
      end
      else
        Scores[i] := SENTINAL_NOT_LEGAL_MOVE;
    end; {-- for --}
  end
  else begin {-- MAX Node --}
    for Temp := 1 to COLS do begin
      i := GetNextBestColumn(Temp, i);
      if IsMoveLegal(theMinimaxBoard, i) then begin
        IncMinimaxEvals;
        theMinimaxBoard^.WhoseTurn := WhoseTurn;
        AddPiece(theMinimaxBoard, i, theRow);
        Scores[i] := GetMinimaxScore(Level + 1, theRow, i, theMove) + ColumnValue(i);
        RemovePiece(theMinimaxBoard, theRow, i);
      end
      else
        Scores[i] := SENTINAL_NOT_LEGAL_MOVE;
    end; {-- for --}
  end; {-- else --}

  {-- Pick the best move that maximizes my play. --}
  if Level mod 2 = 1 then begin
    theLocalScore := MAXINT;
    theMove := 0;
    for Temp := 1 to COLS do begin
      i := GetNextBestColumn(Temp, i);
      if (Scores[i] <> SENTINAL_NOT_LEGAL_MOVE) and
           (Scores[i] < theLocalScore) then begin
        theLocalScore := Scores[i];
        theMove := i;
      end;
    end;
  end
  else begin
    theLocalScore := -MAXINT;
    theMove := 0;
    for Temp := 1 to COLS do begin
```

```
        i := GetNextBestColumn(Temp, i);
        if (Scores[i] <> SENTINAL_NOT_LEGAL_MOVE) and
             (Scores[i] > theLocalScore) then begin
          theLocalScore := Scores[i];
          theMove := i;
        end;
      end;
    end;

    GetMinimaxScore := theLocalScore;
  end; {-- GetMinimaxScore --}
```

The higher we set *theMinimaxDepthBound*, the better Maximilian will do. However, the time for Maximilian to find a solution grows exponentially with *theMinimaxDepthBound*. Figure 5.1 gives a graph corresponding to the time needed to make a move given a depth bound.

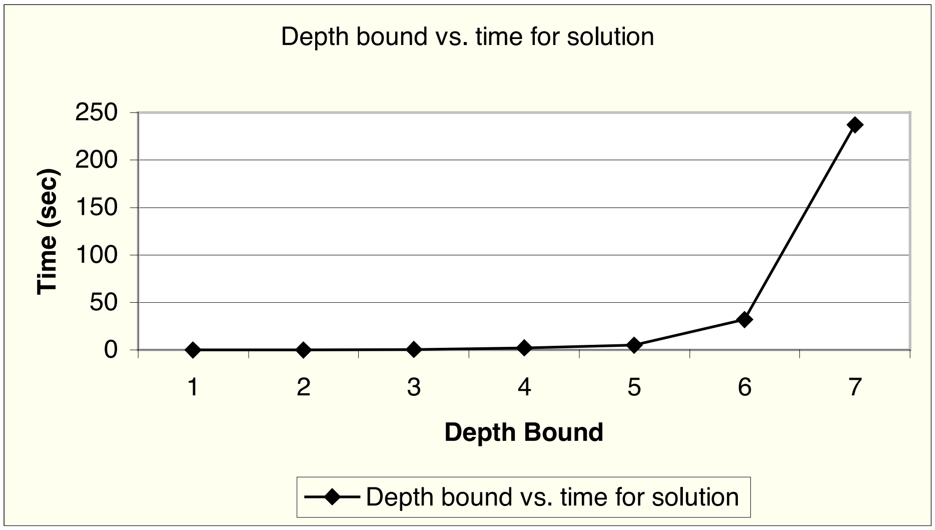

Figure 5.1 – Graph showing the amount of time required to obtain a move at a given depth bound for Maximilian.

5.2. Solomon the Solve Player

The second simplest algorithm that we have implemented is the Solve algorithm. In *Pear*, **Solomon** is the player that uses the Solve algorithm. Here is the Pascal algorithm used:

```
Function GetSolveScore(Level: Integer; Row, Col: Integer; var theMove: Integer): Integer;
  {-----------------------------------------------------------------}
  {-- This function returns the best possible move using the simple --}
  {-- SOLVE algorithm (minimax + check for end game).              --}
  {-- NOTE: Row, Col are the row and column that were moved to to   --}
  {--       get to this state.                                       --}
  {-----------------------------------------------------------------}
  Var
```

```
    i, theLocalScore, theBestMove, theRow, Temp, WhoseTurn: Integer;
    Scores: array[1..Cols] of Integer;
  begin
    {-- Set the WhoseTurn field in the board structure --}
    if Level mod 2 = 1 then WhoseTurn := WhoIsMin
    else WhoseTurn := WhoIsMax;

    {-- If we cannot expand further, return the heuristic (score)   --}
    {-- of the current board.                                       --}
    if level = theSolveDepthBound then begin
      theSolveBoard^.WhoseTurn := WhoIsMax;
      GetSolveScore := h(theSolveBoard);
      Exit;
    end;

    {-- If we have generated a goal state return MAX. --}
    {-- If we have generated a tie state return 0.    --}
    if (Level <> 0) then begin
      theSolveBoard^.WhoseTurn := WhoseTurn;
      if IsGameOver(theSolveBoard, Row, Col, Temp, Temp, Temp) then begin
        if Level mod 2 = 1 then GetSolveScore := HEURISTIC_MAX
        else GetSolveScore := -HEURISTIC_MAX;
        Exit;
      end;
      if IsTieGame(theSolveBoard) then begin
        GetSolveScore := 0;
        Exit;
      end;
    end;

    {-- Get the score of all children's nodes. --}
    if Level mod 2 = 1 then begin  {-- MIN Node --}
      for Temp := 1 to COLS do begin
        i := GetNextBestColumn(Temp, i);
        if IsMoveLegal(theSolveBoard, i) then begin
          IncSolveEvals;
          theSolveBoard^.WhoseTurn := WhoseTurn;
          AddPiece(theSolveBoard, i, theRow);
          Scores[i] := GetSolveScore(Level + 1, theRow, i, theMove) - ColumnValue(i);
          RemovePiece(theSolveBoard, theRow, i);
          if Scores[i] <= -HEURISTIC_MAX then begin
            GetSolveScore := Scores[i];
            theMove := i;
            Exit;
          end;
        end
        else
          Scores[i] := SENTINAL_NOT_LEGAL_MOVE;
      end; {-- for --}
    end
    else begin {-- MAX Node --}
      for Temp := 1 to COLS do begin
        i := GetNextBestColumn(Temp, i);
        if IsMoveLegal(theSolveBoard, i) then begin
          IncSolveEvals;
          theSolveBoard^.WhoseTurn := WhoseTurn;
          AddPiece(theSolveBoard, i, theRow);
          Scores[i] := GetSolveScore(Level + 1, theRow, i, theMove) + ColumnValue(i);
          RemovePiece(theSolveBoard, theRow, i);
          if Scores[i] >= HEURISTIC_MAX then begin
            GetSolveScore := Scores[i];
            theMove := i;
            Exit;
          end;
        end
      else
        Scores[i] := SENTINAL_NOT_LEGAL_MOVE;
    end; {-- for --}
  end; {-- else --}

  {-- Pick the best move that maximizes my play. --}
  if Level mod 2 = 1 then begin
    theLocalScore := MAXINT;
    theMove := 0;
    for Temp := 1 to COLS do begin
      i := GetNextBestColumn(Temp, i);
      if (Scores[i] <> SENTINAL_NOT_LEGAL_MOVE) and
           (Scores[i] < theLocalScore) then begin
        theLocalScore := Scores[i];
        theMove := i;
      end;
    end;
  end
  else begin
    theLocalScore := -MAXINT;
    theMove := 0;
    for Temp := 1 to COLS do begin
      i := GetNextBestColumn(Temp, i);
      if (Scores[i] <> SENTINAL_NOT_LEGAL_MOVE) and
           (Scores[i] > theLocalScore) then begin
        theLocalScore := Scores[i];
        theMove := i;
      end;
    end;
  end;

  GetSolveScore := theLocalScore;
end; {-- GetSolveScore --}
```

As described in section 2.2, the Solve algorithm does a little better than MiniMax since it does not expand any nodes once it has found a winning or losing node. Figure 5.2 gives a graph depicting the depth bound relative to the time to find an answer.

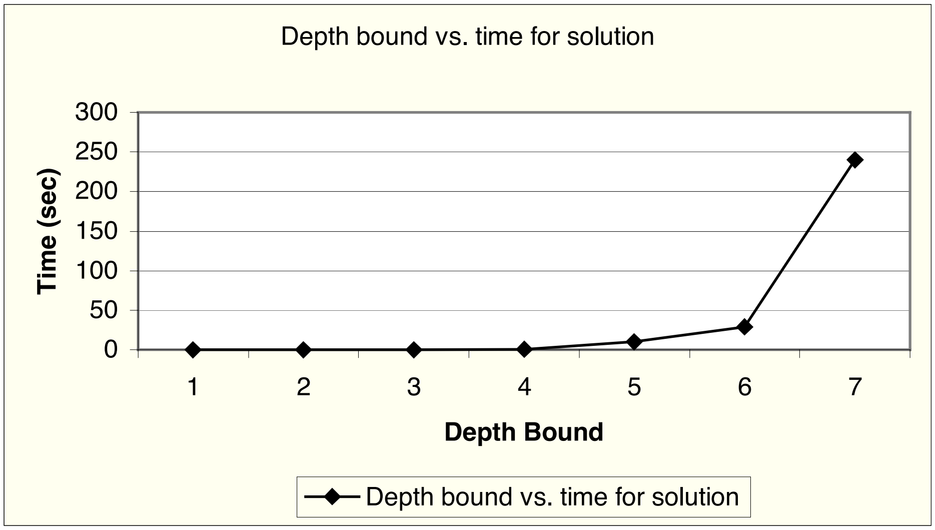

Figure 5.2 – Graph showing the amount of time required to obtain a move at a given depth bound for Solve.

5.3. Annabelle the Alpha-Beta Player

Sam definitely shows some potential. However, he is a very simple game player. The third player, **Annabelle**, uses the most commonly implemented game-playing algorithm Alpha-Beta Pruning. Here is the algorithm as implemented in Pascal:

```
Function GetABScore(A, B, Level, Row, Col: Integer; var theMove: Integer): Integer;
  {---------------------------------------------------------------------}
  {-- This function returns the best possible move using the Alpha- --}
  {-- Beta algorithm.                                                --}
  {-- NOTE: Row, Col are the row and column that were moved to to   --}
  {--       get to this state.                                       --}
  {---------------------------------------------------------------------}
  Var
    i, theLocalScore, theBestMove, theRow, Temp, WhoseTurn: Integer;
    Scores: array[1..Cols] of Integer;
    Alpha, Beta: Integer;
  begin
    {-- Set the WhoseTurn field in the board structure --}
    if Level mod 2 = 1 then WhoseTurn := WhoIsMin
    else WhoseTurn := WhoIsMax;

    {-- If we cannot expand further, return the heuristic (score)   --}
    {-- of the current board.                                        --}
    if level = theABDepthBound then begin
      theABBoard^.WhoseTurn := WhoIsMax;
      GetABScore := h(theABBoard);
      Exit;
    end;

    {-- If we have generated a goal state return MAX. --}
    {-- If we have generated a tie state return 0.    --}
    if (Level <> 0) then begin
      theABBoard^.WhoseTurn := WhoseTurn;
      if IsGameOver(theABBoard, Row, Col, Temp, Temp, Temp) then begin
```

```
                if Level mod 2 = 1 then GetABScore := HEURISTIC_MAX
                else GetABScore := -HEURISTIC_MAX;
                Exit;
              end;
              if IsTieGame(theABBoard) then begin
                GetABScore := 0;
                Exit;
              end;
            end;

            {-- Initialize the alpha and beta values. --}
            Alpha := A;
            Beta := B;

            {-- Get the score of all children's nodes. --}
            if Level mod 2 = 1 then begin {-- a MIN node --}
              for Temp := 1 to COLS do begin
                i := GetNextBestColumn(Temp, i);
                if IsMoveLegal(theABBoard, i) then begin
                  IncABEvals;
                  theABBoard^.WhoseTurn := WhoseTurn;
                  AddPiece(theABBoard, i, theRow);
                  Scores[i] := GetABScore(Alpha, Beta, Level + 1, theRow, i, theMove) -
ColumnValue(i);
                  RemovePiece(theABBoard, theRow, i);
                  Beta := Minimum(Beta, Scores[i]);
                  if Alpha >= Beta then begin
                    GetABScore := Alpha - 4;
                    theMove := i;
                    Exit;
                  end;
                  if Scores[i] <= -HEURISTIC_MAX then begin
                    GetABScore := Scores[i];
                    theMove := i;
                    Exit;
                  end;
                end
                else
                  Scores[i] := SENTINAL_NOT_LEGAL_MOVE;
              end; {-- for --}
            end
            else begin {-- MAX node --}
              for Temp := 1 to COLS do begin
                i := GetNextBestColumn(Temp, i);
                if IsMoveLegal(theABBoard, i) then begin
                  IncABEvals;
                  theABBoard^.WhoseTurn := WhoseTurn;
                  AddPiece(theABBoard, i, theRow);
                  Scores[i] := GetABScore(Alpha, Beta, Level + 1, theRow, i, theMove) +
ColumnValue(i);
                  RemovePiece(theABBoard, theRow, i);
                  Alpha := Maximum(Alpha, Scores[i]);
                  if Alpha >= Beta then begin
                    GetABScore := Beta + 4;
                    theMove := i;
                    Exit;
                  end;
                  if Scores[i] >= HEURISTIC_MAX then begin
                    GetABScore := Scores[i];
                    theMove := i;
                    Exit;
                  end;
                end
                else
                  Scores[i] := SENTINAL_NOT_LEGAL_MOVE;
```

```
    end; {-- for --}
  end; {-- else --}

  {-- Pick the best move that maximizes my play. --}
  if Level mod 2 = 1 then begin
    theLocalScore := MAXINT;
    theMove := 0;
    for Temp := 1 to COLS do begin
      i := GetNextBestColumn(Temp, i);
      if (Scores[i] <> SENTINAL_NOT_LEGAL_MOVE) and
           (Scores[i] < theLocalScore) then begin
        theLocalScore := Scores[i];
        theMove := i;
      end;
    end;
  end
  else begin
    theLocalScore := -MAXINT;
    theMove := 0;
    for Temp := 1 to COLS do begin
      i := GetNextBestColumn(Temp, i);
      if (Scores[i] <> SENTINAL_NOT_LEGAL_MOVE) and
           (Scores[i] > theLocalScore) then begin
        theLocalScore := Scores[i];
        theMove := i;
      end;
    end;
  end;

  GetABScore := theLocalScore;
end; {-- GetSolveScore --}
```

This algorithm is not much more complicated than MiniMax and Solve. However, we end up reducing an enormous amount of expanded nodes. Figure 5.3 gives a graph corresponding to the average time needed to make a move given a depth bound using α-β.

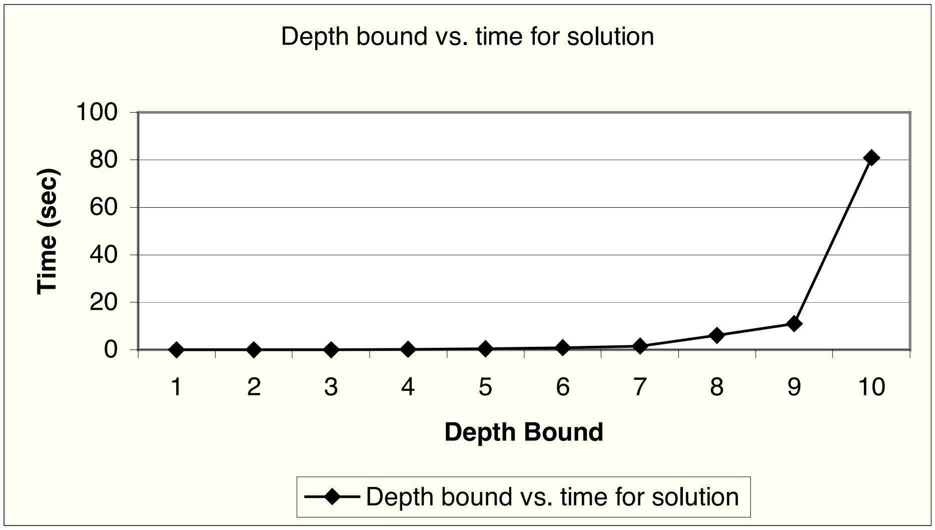

Figure 5.3 – Graph showing the average amount of time required to obtain a move at a given depth bound for α-β.

5.4. Scott the Scout Player

The most complicated automated player implemented in *Pear* uses the Scout algorithm. **Scott** is the name of our Scout algorithm in *Pear*. Here is the description of both the *Test* and the *Eval* function as implemented in Pascal:

```
Function Test(Level, v, Row, Col: Integer) : Boolean;
  {---------------------------------------------------------------}
  {-- Returns true if estimated value of the current board > v.     --}
  {-- Row and Col are the last move that got us here.               --}
  {---------------------------------------------------------------}
  Var
    i, WhoseTurn, Temp, theRow: Integer;
    TestResult: Boolean;
  begin
    if Level mod 2 = 1 then WhoseTurn := WhoIsMin
    else WhoseTurn := WhoIsMax;

    {-- If we cannot expand further, check the heuristic (score)    --}
    {-- of the current board with v.                                --}
    if (level = theScoutDepthBound) then begin
      theScoutBoard^.WhoseTurn := WhoIsMax;
      if h(theScoutBoard) > v then Test := True
      else Test := False;
      Exit;
    end;
    if (Level <> 0) then
      if IsGameOver(theScoutBoard, Row, Col, Temp, Temp, Temp) then begin
        if Level mod 2 = 1 then Test := True
        else Test := False;
        Exit;
      end
      else if IsTieGame(theScoutBoard) then begin
        if 0 > v then Test := True
      else Test := False;
      Exit;
    end;
  {-- If we are at a MAX node --}
  if Level mod 2 = 0 then begin
    for Temp := 1 to COLS do begin
      i := GetNextBestColumn(Temp, i);
      if IsMoveLegal(theScoutBoard, i) then begin
        IncScoutTests;
        theScoutBoard^.WhoseTurn := WhoseTurn;
        AddPiece(theScoutBoard, i, theRow);
        TestResult := Test(Level + 1, v, theRow, i);
        RemovePiece(theScoutBoard, theRow, i);
        if TestResult then begin
          Test := True;
          Exit;
        end; {-- if --}
      end; {-- if --}
    end; {-- for --}
    Test := False;
    Exit;
  end
  else begin  {-- MIN node --}
    for Temp := 1 to COLS do begin
      i := GetNextBestColumn(Temp, i);
      if IsMoveLegal(theScoutBoard, i) then begin
        IncScoutTests;
        theScoutBoard^.WhoseTurn := WhoseTurn;
        AddPiece(theScoutBoard, i, theRow);
        TestResult := Test(Level + 1, v, theRow, i);
        RemovePiece(theScoutBoard, theRow, i);
        if not TestResult then begin
          Test := False;
          Exit;
        end; {-- if --}
      end; {-- if --}
    end; {-- for --}
    Test := True;
    Exit;
  end; {-- else --}
end; {-- Test --}

Function Eval(Level, Row, Col: Integer; var theMove: Integer) : Integer;
{-----------------------------------------------------------------}
{-- This function returns the best possible move using the        --}
{-- SCOUT algorithm.                                              --}
{-- NOTE: Row, Col are the row and column that were moved to to   --}
{--       get to this state.                                      --}
{-----------------------------------------------------------------}
Var
  Value, i, theLocalScore, theBestMove, theRow, Temp, WhoseTurn: Integer;
  Scores: array[1..Cols] of Integer;
  GotValue, TestResult: Boolean;
begin
  {-- Set the WhoseTurn field in the board structure --}
  if Level mod 2 = 1 then WhoseTurn := WhoIsMin
  else WhoseTurn := WhoIsMax;

  {-- If we cannot expand further, return the heuristic (score)   --}
  {-- of the current board.                                       --}
  if level = theScoutDepthBound then begin
    theScoutBoard^.WhoseTurn := WhoIsMax;
    Eval := h(theScoutBoard);
    Exit;
  end;
```

```
{-- If we have generated a goal state return MAX. --}
{-- If we have generated a tie state return 0.    --}
if (Level <> 0) then begin
  theScoutBoard^.WhoseTurn := WhoseTurn;
  if IsGameOver(theScoutBoard, Row, Col, Temp, Temp, Temp) then begin
    if Level mod 2 = 1 then Eval := HEURISTIC_MAX
    else Eval := -HEURISTIC_MAX;
    Exit;
  end;
  if IsTieGame(theScoutBoard) then begin
    Eval := 0;
    Exit;
  end;
end;

{-- If we are at a MAX node --}
if Level mod 2 = 0 then begin
  {-- Get the estimated value of the first legal move --}
  Temp := 0;
  GotValue := False;
  repeat
    Inc(Temp);
    i := GetNextBestColumn(Temp, i);
    if IsMoveLegal(theScoutBoard, i) then begin
      GotValue := True;
      theBestMove := i;
      IncScoutEvals;
      theScoutBoard^.WhoseTurn := WhoseTurn;
      AddPiece(theScoutBoard, i, theRow);
      Value := Eval(Level + 1, theRow, i, theMove) + ColumnValue(i);
      RemovePiece(theScoutBoard, theRow, i);
    end;
  until GotValue;

  {-- Check if any better moves --}
  for Temp := Temp + 1 to COLS do begin
    i := GetNextBestColumn(Temp, i);
    if IsMoveLegal(theScoutBoard, i) then begin
      IncScoutTests;
      theSCoutBoard^.WhoseTurn := WhoseTurn;
      AddPiece(theScoutBoard, i, theRow);
      TestResult := Test(Level + 1, Value, theRow, i);
      if TestResult then begin
        IncScoutEvals;
        Value := Eval(Level + 1, theRow, i, theMove) + ColumnValue(i);
        theBestMove := i;
      end;
      RemovePiece(theScoutBoard, theRow, i);
    end;
  end; {-- for --}
end
else begin {-- MIN node --}
  {-- Get the estimated value of the first legal move --}
  Temp := 0;
  GotValue := False;
  repeat
    Inc(Temp);
    i := GetNextBestColumn(Temp, i);
    if IsMoveLegal(theScoutBoard, i) then begin
      GotValue := True;
      theBestMove := i;
      IncScoutEvals;
      theScoutBoard^.WhoseTurn := WhoseTurn;
      AddPiece(theScoutBoard, i, theRow);
```

```
          Value := Eval(Level + 1, theRow, i, theMove) - ColumnValue(i);
          RemovePiece(theScoutBoard, theRow, i);
        end;
      until GotValue;

      {-- Check if any better moves --}
      for Temp := Temp + 1 to COLS do begin
        i := GetNextBestColumn(Temp, i);
        if IsMoveLegal(theScoutBoard, i) then begin
          IncScoutTests;
          theSCoutBoard^.WhoseTurn := WhoseTurn;
          AddPiece(theScoutBoard, i, theRow);
          TestResult := Test(Level + 1, Value, theRow, i);
          if not TestResult then begin
            IncScoutEvals;
            Value := Eval(Level + 1, theRow, i, theMove) - ColumnValue(i);
            theBestMove := i;
          end;
          RemovePiece(theScoutBoard, theRow, i);
        end;
      end; {-- for --}
    end; {-- else --}
    Eval := Value;
    theMove := theBestMove;
  end; {-- Eval --}
```

Figure 5.4 describes the average time it takes for Scott to return a move given different depth bounds. Its graph is very similar to the one for Alpha-Beta Pruning.

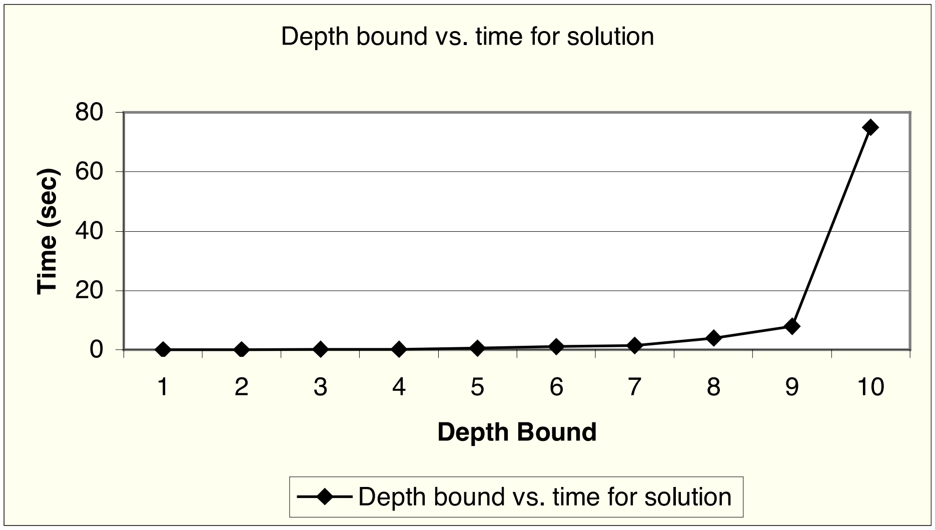

Figure 5.4 – Graph showing the average amount of time required to obtain a move at a given depth bound for Scout.

6. Results In *Pear*

In this section, I will discuss the results of several tests on the different adversary searches implemented in *Pear*. The basic idea is to verify if the theoretical analysis done in sections 2 and 3 hold in practice. First, I will compare the average amount of time needed to obtain a solution. Secondly, I will discuss the amount of nodes expanded by each automated player. Finally, the results of a tournament between the players will be shown.

6.1. Time Complexity

In order to compare the time complexities of the algorithms, I simply observed and noted the average time it took for a particular algorithm to return a move. Figure 6.1 shows this information using a graphical representation.

As the theory predicted, Maximilian and Solomon are both very close to each other. Solomon does in fact expand a little fewer nodes than Maximilian (as will be shown in section 6.2) however, both algorithms have the same theoretical time complexity. This graph shows this clearly. I could not observe times for depths greater than seven for both of these algorithms since these times would have been very long. The graph shows all four players' algorithms growing exponentially with increased depth bounds.

In section 4.1, I showed that at the beginning of a game, an automated player would have to expand 42 plies in order to search the whole game tree. By the graph in Figure 6.1, this would be obviously impossible. Just checking more than 7 plies with Maximilian and Solomon requires an unreasonable amount of time.

Similarly, both Annabelle and Scott matched closely their theoretical analysis. That is, both have similar graphs supporting the fact that they have the same time complexity. Notice also that both of them do much better than Maximilian and Solomon. This is due to the fact that they expand substantially fewer nodes. Thus, they can look deeper in the tree for a solution.

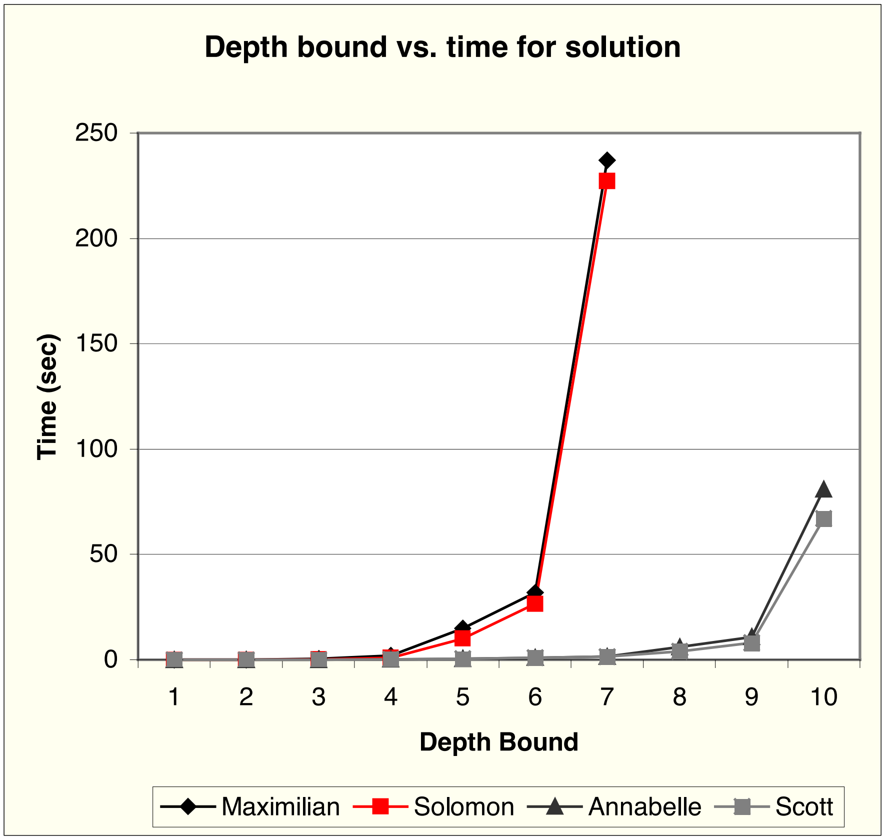

Figure 6.1 – Graph showing the average time for different computer players to return a solution for a given depth bound.

From this, it should be clear that generally Annabelle and Scott would beat Maximilian and Solomon if all algorithms were restricted to return a solution in a certain time period. This way, Annabelle and Scott could look deeper in the tree than Maximilian and Solomon and thus return better moves. The tournament in section 6.2 will demonstrate this result.

6.2. Expanded Nodes

As for the number of nodes expanded, we have to normalise the comparison by forcing a static depth bound for each player. For the sake of speed, I have chosen to limit all the algorithms to a depth of five plies. Through testing in *Pear*, I obtained the results shown in Figure 6.2. The number of nodes expanded was averaged over 70 games. Every automated player played every other automated player 20 times, ten times as Player A and ten times as Player B. The remaining ten games where played against itself (e.g. Annabelle vs. Annabelle). Thus, in total, 160 games were played.

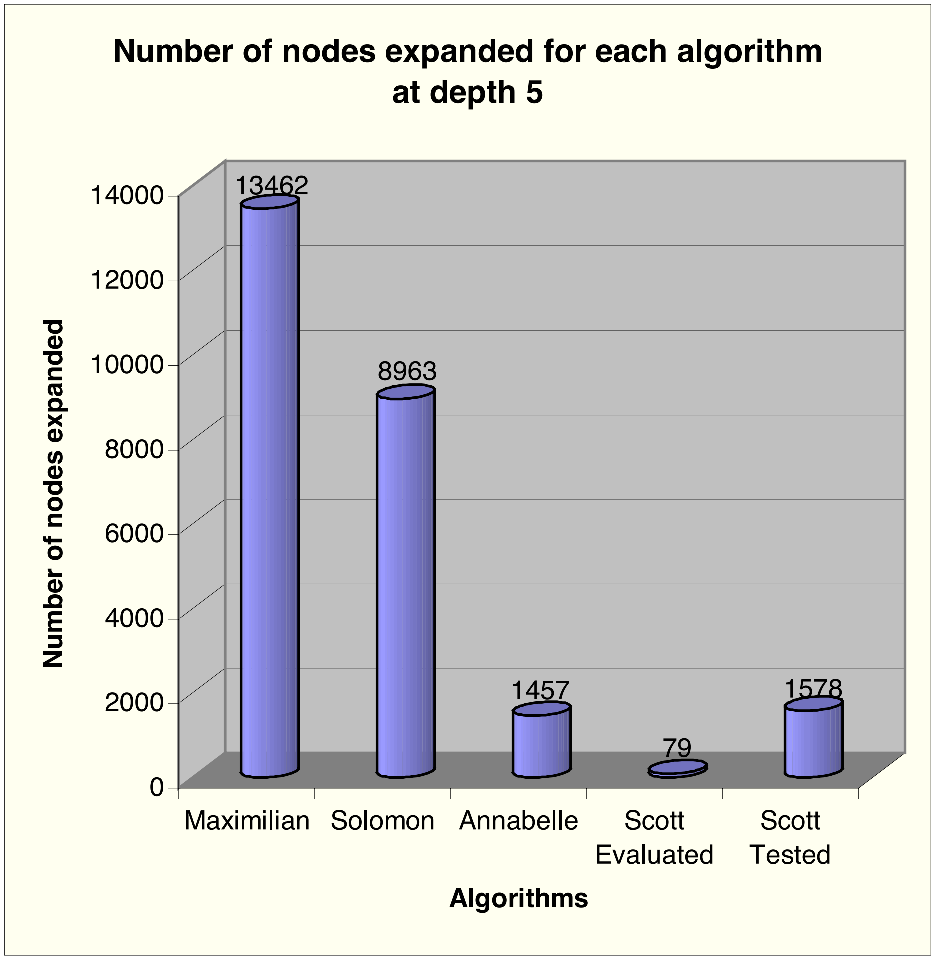

Figure 6.2 – Each bar represents the number of nodes expanded by each algorithm at a depth of five plies.

As the theory predicted, Annabelle and Scott expanded much fewer nodes than Maximilian and Solomon. Annabelle expanded almost ten times fewer nodes than Maximilian! Also, remember that both algorithms are optimal. Therefore, Maximilian expands an extra 12,000 nodes that it does not need. It would be much wiser to expand the same number of nodes over more plies.

The interesting information in Figure 6.2 is the number of nodes expanded by Scott. By using the Scout algorithm, he was able to expand only 79 nodes! That is incredible in comparison to Maximilian. What is important to notice is that Scott **tested** 1578 nodes. It would appear that this algorithm is a little worse than Annabelle's. In fact, it turns out that Scott is quicker since his *Test* function is much quicker than the evaluation function (see Section 3.2).

Interestingly enough, of the 1578 nodes that Scott tested, only 79 nodes needed to be expanded (i.e. passed the Test function). This confirms Pearl's suggestion that most of the time the Scout algorithm does not need to expand a branch (see Section 3.2). This suggests that at any point in time, the algorithm must only consider a small subset of all the possible moves.

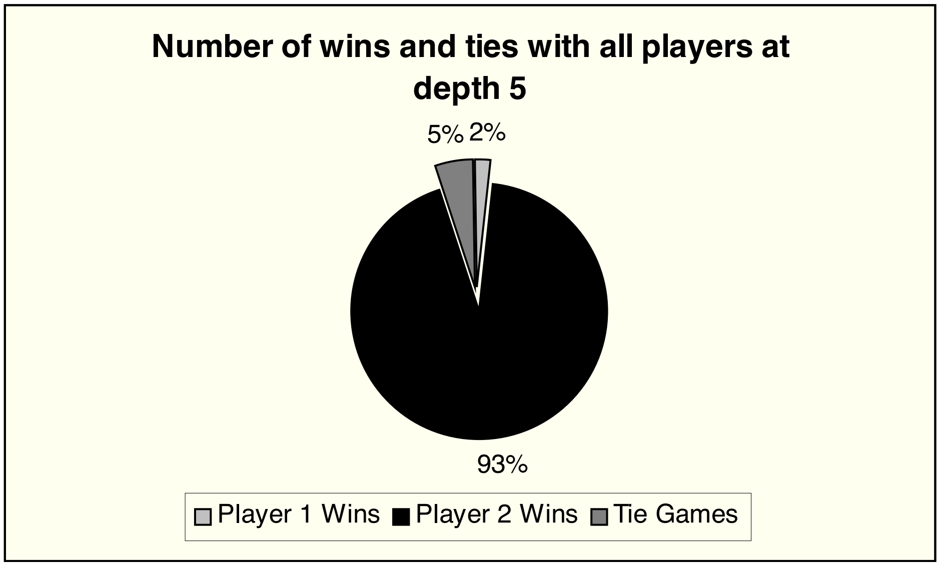

Figure 6.3 – Percentages of wins and ties for 160 games with all players playing with a depth bound of five.

An unexpected result from this experiment showed that it is advantageous for a player to be Player 2 in *Pear*. Figure 6.3 shows that 93% of the 160 games played had Player 2 winning the game! Only 5% of the time we had a tie game and only 2% of the time Player 1 won. Therefore, contrary to what we would imagine, it is better to be Player 2 in *Pear*!

Of course this is not true for every game. In fact, in tic-tac-toe (see section 1.3.1), it is favorable to be Player 1. [Wang, 1965] wrote a very interesting article showing that in some games it is advantageous to be Player 1, some Player 2, and some it does not matter at all.

6.3. Tournament

In this section, I present the results of a **tournament** between the automated players in *Pear*: Maximilian, Solomon, Annabelle, and Scott. I will begin by explaining the rules that were used to create the tournament. Then, I will provide the reader with the results.

6.3.1. Rules

The setup of the tournament is simple. Each player will play **twenty** games against every other player. Ten of those twenty games will be played as player A and the other ten as player B. The reason for this should be clear from section 6.2 (i.e. it is favourable to start as Player 2).

In order to evaluate the players, we must award them scores. Two points will be awarded for every win, one point for every tie, and zero points for every loss. The winner of the tournament will be the player with the most points. Since each player will play 60 games, the highest possible score would be 120 (i.e. the player won every game). The lowest score possible is 0 (i.e. the player lost every game). An average player is expected to score $120 \div 2 = 60$ points.

To simplify the implementation, I have selected a static depth bound for each player. Our criterion for this selection is that we want each player to return a move within approximately ten seconds. Thus, Maximilian will play at depth 5, Solomon at depth 5, Annabelle at depth 8, and Scott at depth 9. These choices were manually composed by selecting a depth bound for each player with respect to the ten second average. Because of these depth bounds and the fact that all algorithms are optimal, we expect Scott to be the champion, followed by Annabelle in second place, and Maximilian and Solomon fighting for third.

Note that the theory suggested that α-β and Scout should be able to expand twice as many levels as MiniMax. Unfortunately, I was unable to achieve exactly this in *Pear* as theory described. The reason that we cannot is due to two reasons. First, we assumed that, in *Pear*, we had a constant branching of seven. However, this is not completely true. In fact, once a column has been completely filled, the branching factor becomes six. So, the average branching factor would be greater than six and approaching seven but not equal to seven. Secondly, we should have reversed the order in which moves were examined when we were finding the score of a MIN node as opposed to a MAX node. Thus, we could have eliminated more paths by finding the best expected move first for both MAX and MIN nodes.

6.3.2. Results

Here I present how the players performed in the tournament. Table 6.1 shows the results of every game. Each square represents the number of wins/losses/ties from respect to the player on the row (vs. the player on the column). The last column shows the total number of wins/losses/ties for each player.

	Maximilian	**Solomon**	**Annabelle**	**Scott**	**Total**
Maximilian	N/A	10 wins 10 losses 0 ties	0 wins 20 losses 0 ties	0 wins 20 losses 0 ties	10 wins 50 losses 0 ties
Solomon	10 wins 10 losses 0 ties	N/A	0 wins 20 losses 0 ties	0 wins 20 losses 0 ties	10 wins 50 losses 0 ties
Annabelle	20 wins 0 losses 0 ties	20 wins 0 losses 0 ties	N/A	1 win 11 losses 8 ties	41 wins 11 losses 8 ties
Scott	20 wins 0 losses 0 ties	20 wins 0 losses 0 ties	11 wins 1 loss 8 ties	N/A	51 wins 1 loss 8 ties

Table 6.1 – Results of the tournament. Every player played twenty games against every other player.

Clearly, Scott was the champion of the tournament. Figure 6.4 shows the total scores awarded to each player. Note that 120 points was the maximum score possible. Scott led the field with 110 points losing only one game and tying eight games to Annabelle. In second place, with a very good score, Annabelle recorded 90 points. She lost only 11 matches and tied eight against Scott but won every game against Maximilian and Solomon. Maximilian and Solomon were tied for last place. Both of them lost all their games against Annabelle and Scott. Between themselves, they split the games ten apiece. As you might have guessed, based on the prediction in section 6.2, they all won the games when they were Player 2.

These results were exactly as we expected. The player with the highest depth bound won and the players with the lowest depth bound lost. Since Annabelle and Scott searched much deeper than Maximilian and Solomon, they both did not lose a game against them. However, since Scott searched only one ply more than Annabelle, their scores were much closer. Eight times, Annabelle was able to get a draw. She was even able to win a game on one occasion.

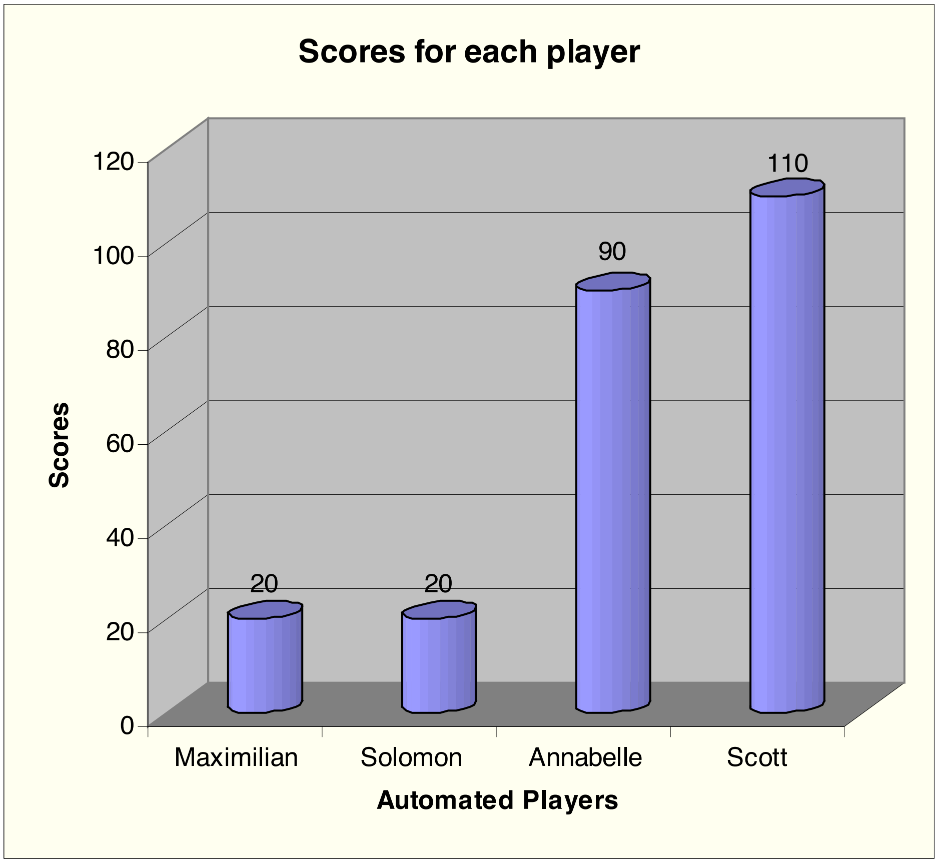

Figure 6.4 – Total score for each automated player in *Pear*. Each player played 60 games; 2 points for a win, 1 point for a tie.

In conjunction with our discussion in section 6.2, it is interesting to note that once again we have proof that Player 2 has an advantage in *Pear*. In the games between Annabelle and Scott, every game where Scott was Player 2, he won. Similarly, every time Annabelle got a draw or a win, she was playing as Player 2. Thus, it is reasonable to assume that Player 2 has a slight advantage over Player 1.

To recap, the least nodes an adversary search algorithm expands, the deeper the game tree may be searched. As shown in *Pear*, the deeper an algorithm may search, the more successful it will be. This is, of course, assuming that all the algorithms are optimal. Thus, our practical experiments support the theory presented in sections 2 and 3.

7. Conclusion

There exists many different types of adversary search algorithms. We have examined two kinds: depth-first search algorithms and best-first search algorithms. The success of an algorithm is based both on its ability to return the best possible solution (optimality) and on doing so by expanding the fewest nodes possible. All algorithms introduced in this paper were optimal algorithms and therefore the better ones were the ones that expanded the fewest nodes.

Any interesting game has a search space that could never be fully searched. Therefore, we showed that a particular board configuration must somehow be evaluated. Such board evaluation functions are domain specific to each individual game. However, typically a good strategy is to assign scores to particular pieces or board positions.

Once we have a board evaluation function, we may use adversary search algorithms to retrieve the best possible move for any player at any point in time. The MiniMax algorithm is the simplest of such algorithms since it does not avoid any search. That is, it expands every node it encounters.

In order to have successful automated players, we realised that our adversary searches must be able to reduce the amount of nodes expanded. The first of such algorithms that we discussed was the Solve algorithm. It acts exactly as MiniMax except when it finds a move that is unbeatable. In such a case, it will automatically return that value without searching for any more moves. Unfortunately, this method reduces hardly any search. Thus, Solve has the same time complexity as MiniMax.

Several advanced adversary search algorithms were then described. We saw that Alpha-Beta Pruning has a time complexity in the order of the square root of MiniMax's complexity. That is a drastic improvement. Theoretically, Alpha-Beta Pruning could produce a move in the same amount of time as MiniMax and Solve by searching through twice the number of plies. Thus, Alpha-Beta Pruning is a much better algorithm. We then introduced Scout, which is an algorithm that has the same complexity as Alpha-Beta Pruning (or very close to the same complexity). It has been shown however, in some cases, to outperform Alpha-Beta.

We then described briefly a few best-first search algorithms. The first that we introduced was SSS*. It theoretically outperforms α-β. However, SSS* suffers from the poor space complexity of best-first search algorithms. Therefore, although it is of theoretical interest, it is impractical. By combining transposition tables and a modification of Pearl's *Test* function in Scout, we were able to solve the space complexity problem. These ideas became known as the MT algorithms. A general driver can be built from MT and used to create a variety of MT-based adversary searches. The two most interesting were MTD-∞ which was shown to be equivalent to SSS* and MTD-*f* which outperforms all previously discussed algorithms.

We put the theoretical analysis of the above depth-first searches to a practical test using a game called *Pear* (based on the game Connect four). As predicted by their theoretical analysis, Scout was the most successful followed by α-β, Solve, and MiniMax respectively. However, we failed to achieve the average complexity for α-β and Scout. This was mainly due to the static ordering of the operations examined by the players. However, they came very close to their theoretical expectations. In the end, they were able to evaluate many more plies that MiniMax and Solve.

We have seen that the best adversary searches return a good solution by expanding few nodes. However, we restricted this paper to a very specific type of game, namely two-person perfect information zero-sum games. If we were to consider multi-player non-zero sum games, we would need to consider the case where it might be beneficial for a player to select a sub-optimal move to gain the trust of another player. Also, allowing negotiations between players during the game would drastically change the strategies discussed in this paper. Moreover, if added incomplete information, the player would have to deal with risks (i.e. chance). In such games, the search space grows even faster since at a chance node, we must consider all outcomes and their probabilities. Clearly, there is much more to games than the class of two-person perfect information zero-sum games.

References

[AAAI, 1996] American Association for Artificial Intelligence, Searching Game Trees Under Memory Constraints, *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, Oregon, 1996.

[Berlekamp, 1982] Berlekamp, E.R., et. al. *Winning Ways, For Your Mathematical Plays Vol.1*, Academic Press, London; New York, 1986.

[Hofstadter, 1985] Hofstadter, Douglas R., *Metamagical Themas*, Basic Books Inc., United States, 1985.

[Johnson, 1997] Massey, Martin, *Alpha-Beta Pruning*, http://cs-alb-pc3.massey.ac.nz/notes/59318/l4.html, Massey University, Albany, 1997.

[Rapoport, 1966] Rapoport, Anatol, *Two-Person Game Theory*, The University Of Michigan Press, The University Of Michigan, 1966.

[Pearl, 1984] Pearl, *Heuristics: Intelligent Search Strategies For Computer Problem Solving*, 1984.

[Plaat, 1994] Plaat, Aske, et al, *A New Paradigm for Minimax Search*, The University Of Alberta, 1994.

[Plaat, 1995] Plaat, Aske, et al, *Best-First Fixed-Depth Minimax Algorithms*, The University Of Alberta, 1995.

[Pijls, 1995] Pijls, W. and de Bruin, A., *Another view on the* SSS^* *algorithm*, Erasmus University Rotterdam, http://www.cs.few.eur.nl/few/inf/publicaties/rapporten/eur-few-cs-90-01.html, 1995.

[Roizen, 1983] Roizen, I and Pearl, J., A Minimax Algorithm Better than Alpha-Beta? Yes and No., *Artificial Intelligence 21*, 1983.

[Russell, 1995] Russell, Stuart J. and Norvig, Peter, *Artificial Intelligence: A Modern Approach*, Prentice-Hall Inc., New Jersey, 1995.

[Stockman, 1979] Stockman, G., A minimax algorithm better than alpha-beta? *Artificial Intelligence 12*, 1979.

[UGAI, 1994] UGAI Workshop, *MiniMax with Alpha-Beta Cutoff*, http://yoda.cis.temple.edu:8080/UGAIWWW/lectures97/search/minimax/index.html, 1994.

[Von Newmann, 1944] Von Newmann, J. and Morgenstern, Oi, *The Theory of Games and Economic Behaviour – First Edition*, Princeton University Press, New Jersey.

[Wang, 1965] Wang, Hao, Scientific American Volume 213 Number 5, *Games, Logic and Computers*, 1965.

Index

#

α-β ... *See* Alpha-Beta Pruning

A

advanced adversary search algorithms ... 9
adversary search algorithms ... 5
Alpha-Beta Pruning ... 9, 15, 16, 18, 20, 34
asymmetry ... 18

B

best-first search ... 16
board evaluation function ... 7, 27
  symmetry ... 27

C

children ... 5
cluster ... 16
complete payoff ... 2
complexity
  Alpha-Beta Pruning ... 11
  games ... 2
  MiniMax ... 7
  Pear ... *See* Pear complexity
  Scout ... 15
  Solve ... 8
configurations ... 3
Connect Four ... *See* Pear
cut-offs ... 13

D

depth bound ... 6
  dynamic ... 6
  static ... 6
depth-first searches ... 5

E

effective branching factor ... 11
estimated board value ... *See* board evaluation function
estimation of quietness ... *See* quiescence
exponential growth ... *See* intractable

F

frontier node ... 6

G

game ... 1
game theory ... 1
graphical user interface ... 23

H

hard ... *See* intractable
horizon problem ... 6

I

IDSSS* ... *See* Iterative Deepening SSS*
initial state ... 3
interface ... 23
intractable ... 2, 7
iterative deepening ... 17
Iterative Deepening SSS* ... 17

L

leaf node ... *See* frontier node

M

maximize ply ... 5
Memory-enhanced Test ... *See* MT
MemSSS* ... 17
MiniMax ... 5, 29
minimize ply ... 5
move ... 1
MT ... 18
  MTD ... 20
  MTD+∞ ... 19
  MTD-*f* ... 20
MT-based Algorithms ... 17

N

negotiation ... 2

O

object oriented ... 23
optimal search ... 5

P

pattern recognition ... 3
Pear ... 23
  Annabelle ... 34
  board evaluation ... 27
  complexity ... 24
  defensive strategies ... 25
  expanded nodes ... 42
  intelligent players ... 29
  legal move ... 23
  Maximilian ... 29
  offensive strategies ... 25
  Player 2 advantage ... 44
  Rules ... 23
  Scott ... 37

  Solomon 31
  strategy 24
  time complexity 41
perfect information 1
ply...1
priority queue 16

Q

quiescence 6
quiet configuration 6

R

reasonable search time..6
recursive algorithm 5
reduce search 3

S

Scout...12, 20, 37
  Eval function 12, 37
  Test function 12, 37
search space 2
SMA* 17
Solve 7, 31
space complexity 16
SSS* 16, 18, 19, 20
symmetrical boards..3

T

termination function 5
tic-tac-toe 3
tournament...44
  results...46
  rules 45
transposition table..18
tree 5
two-person perfect information zero-sum games2